Introduction

Important submission note: For full credit,

- Submit with Phases 1-2 complete by **Monday, March 5** (worth 1 pt).
- Submit with Phases 3-5 complete by **Thursday, March 8**.

You may work with one other partner for the entire project. You will get an extra credit point for submitting the entire project by Wednesday, March 7.

In this project, you will create a tower defense game called Ants Vs. SomeBees. As the ant queen, you populate your colony with the bravest ants you can muster. Your ants must protect their queen from the evil bees that invade your territory. Irritate the bees enough by throwing leaves at them, and they will be vanquished. Fail to pester the airborne intruders adequately, and your queen will succumb to the bees' wrath. This game is inspired by PopCap Games' Plants Vs. Zombies.

This project combines functional and object-oriented programming paradigms, focusing on the material from Chapter 2.5 of Composing Programs. The project also involves understanding, extending, and testing a large program.

Download starter files

The ants.zip archive contains several files, but all of your changes will be made to `ants.py`.

- `ants.py`: The game logic of Ants Vs. SomeBees
- `ants_gui.py`: The original GUI for Ants Vs. SomeBees
- `gui.py`: An new GUI for Ants Vs. SomeBees
- `graphics.py`: Utilities for displaying simple two-dimensional animations
- `state.py`: Abstraction for gamestate for `gui.py`
- `utils.py`: Some functions to facilitate the game interface

- `ucb.py`: Utility functions for CS 61A
- `assets`: A directory of images and files used by `gui.py`
- `img`: A directory of images used by ants_gui.py
- `ok`: The autograder
- `proj3.ok`: The `ok` configuration file
- `tests`: A directory of tests used by `ok`
- `mytests.rst`: A space for you to add your custom tests; see section on adding your own tests

Logistics

This is a 11-day project. You may work with one other partner. You should not share your code with students who are not your partner or copy from anyone else's solutions. In the end, you will submit one project for both partners.

Remember that you can earn an additional bonus point by submitting the project at least 24 hours before the deadline.

The project is worth 27 points. 24 points are assigned for correctness, 1 point for submitting Phases 1-2 by the checkpoint date, and 2 points for the overall composition.

You will turn in the following files:

- `ants.py`
- `mytests.rst` (ungraded)

You do not need to modify or turn in any other files to complete the project. To submit the project, run the following command:

```
python3 ok --submit
```

You will be able to view your submissions on the Ok dashboard.

For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do **not** modify any other functions. Doing so may result in your code failing our autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an **autograder** called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to **log in with your Ok account using your web browser**. Please do so. Each time you run `ok`, it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations, but there are two things you should be aware of.

First, some of the test cases are *locked*. To unlock tests, run the following command from your terminal:

```
python3 ok -u
```

This command will start an interactive prompt that looks like:

```
==================================================
==================
Assignment: Ants Vs. SomeBees
Ok, version ...
==================================================
==================


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~
Unlocking tests


At each "? ", type what you would expect the output
```

```
to be.
Type exit() to quit

---------------------------------------------------
------------------
Question 0 > Suite 1 > Case 1
(cases remaining: 1)

>>> Code here
?
```

At the `?`, you can type what you expect the output to be. If you are correct, then this test case will be available the next time you run the autograder.

The idea is to understand *conceptually* what your program should do first, before you start writing any code.

Once you have unlocked some tests and written some code, you can check the correctness of your program using the tests that you have unlocked:

```
python3 ok
```

Most of the time, you will want to focus on a particular question. Use the `-q` option as directed in the problems below.

We recommend that you submit **after you finish each problem**. Only your last submission will be graded. It is also useful for us to have more backups of your code in case you run into a submission issue.

The `tests` folder is used to store autograder tests, so **do not modify it**. You may lose all your unlocking progress if you do. If you need to get a fresh copy, you can download the zip archive and copy it over, but you will need to start unlocking from scratch.

If you do not want us to record a backup of your work or information about your progress, use the `--local`option when invoking `ok`. With this option, no information will be sent to our course servers.

The Game

A game of Ants Vs. SomeBees consists of a series of turns. In each turn, new bees may enter the ant colony. Then, new ants are placed. Finally, all insects (ants, then bees) take individual actions. Bees either try to move toward the end of the tunnel or they sting ants in their way. Ants perform a different action depending on their type, such as throwing leaves at the bees. The game ends either when a bee reaches the ant queen (you lose), or the entire bee flotilla has been vanquished (you win).

Core concepts

The Colony. This is where the game takes place. The colony consists of several *places* that are chained together to form a tunnel where bees can travel through. The colony has some quantity of food that can be expended to deploy ant troops.

Places. A place links to another place to form a tunnel. The player can place a single ant into each place. However, there can be many bees in a single place.

The Hive. This is the place where bees originate. Bees exit the hive to enter the ant colony.

Ants. Ants are the usable troops in the game that the player places into the colony. Each type of ant takes a different action and requires a different amount of food to place. The two most basic ant types are the `HarvesterAnt`, which adds one food to the colony during each turn, and the `ThrowerAnt`, which throws a leaf at a bee each turn. You will be implementing many more.

Bees. Bees are the antogonistic troops in the game that the player must defend the colony from. Each turn, a bee either advances to the next place in the tunnel if no ant is in its way, or it stings the ant in its way. Bees win when at least one bee reaches the end of a tunnel.

Queen Ant: There is one queen ant in the whole colony. She is able to attack bees but she also has a special ability of fortifying the other ant troops. Bees can also win if they destroy the queen ant.

Core classes

The concepts described above each have a corresponding class that encapsulates the logic for that concept. Here is a summary of the main classes involved in this game:

- `AntColony`: Represents the colony and some state information about the game, including how much food is available, how much time has elapsed, where the `QueenAnt` resides, and all the `Place`s in the game.
- `Place`: Represents a single place that holds insects. At most one `Ant` can be in a single place, but there can be many `Bee`s. `Place` objects have an `exit` and an `entrance` which are also places. Bees travel through a tunnel tunnel by moving to a `Place`'s `exit`.
- `Hive`: Represents the hive where `Bee`s start out.
- `Insect`: A superclass for `Ant` and `Bee`. All insects have an `armor` attribute, their remaining health, and a `place` attribute, the `Place` where they are currently located. Each turn, every active `Insect` in the game performs its `action`.
- `Ant`: Represents ants. Each `Ant` subclass has special attributes or a special `action` that distinguish it from other `Ant` types. For example, a `HarvesterAnt` gets food for the colony and a `ThrowerAnt` attacks `Bee`s. Each ant type also has a `food_cost` attribute that indicates how much it costs to deploy one unit of that type of ant.
- `Bee`: Represents bees. Each turn, a bee either moves to the `exit` of its current `Place` if no ant blocks its path, or stings an ant that blocks its path.

Playing the game

The game can be run in two modes: as a text-based game or using a graphical user interface (GUI). The game logic is the same in either case, but the GUI enforces a turn time limit that makes playing the game more exciting. The text-based interface is provided for debugging and development.

The files are separated according to these two modes. `ants.py` knows nothing of graphics or turn time limits.

To start a text-based game, run

```
python3 ants.py
```

To start a graphical game, run

```
python3 gui.py
```

To start an older version of the graphics, run

```
python3 ants_gui.py
```

When you start the graphical version, a new window should appear. In the starter implementation, you have unlimited food and your ants only throw leaves at bees in their current `Place`. Try playing the game anyway! You'll need to place a lot of `ThrowerAnt`s (the second type) in order to keep the bees from reaching your queen.

The game has several options that you will use throughout the project, which you can view with `python3 ants.py --help`.

```
usage: ants.py [-h] [-d DIFFICULTY] [-w] [--food
FOOD]

Play Ants vs. SomeBees

optional arguments:
  -h, --help     show this help message and exit
  -d DIFFICULTY  sets difficulty of game (test/
easy/medium/hard/insane)
  -w, --water    loads a full layout with water
  --food FOOD    number of food to start with when
testing
```

Your own test cases

Adding your own tests is **entirely optional**; you will not lose points if you submit an empty `mytests.rst` file.
We highly recommend that you add your own tests as you work through the project. It's a really helpful way to speed up the debugging process and improve your understanding of the code.

The course staff has also made an instructional video summarizing the information below.

Adding tests

Adding tests is easy. Directly edit the `mytests.rst` file included with the Ants project. We provide a sample structure to start from, but the test format is actually quite flexible. Here are some simple rules to follow:

- Follow standard Python doctest format. This is what we mostly use for the Ok tests, so feel free to use those as examples.
- For the most part, you may use whitespace as you'd like, but we recommend keeping it organized for your own sake.
- Generally, smaller, simpler tests will be more useful than larger, more complex tests.

You may also find our debugging guide helpful. If you're stuck on a particularly tricky `Ok` test case, a good first step would be to break it up into small parts and test them out yourself in `mytests.rst`.

Running tests

To run all your tests in `mytests.rst` with verbose results:

```
python3 ok -t -v
```
If you put your tests in a different file or split your tests up into multiple files:

```
python3 ok -t your_new_filename.rst
```
To run just the tests from suite 1 case 1 in `mytests.rst`:

```
python3 ok -t --suite 1 --case 1
```
You might have noticed that there's a "test coverage" percentage for your tests (note that coverage statistics are only returned when running all tests). This is a measure of your test's code coverage. If you're interested, you can find more information in our reference guide.

Phase 1: Basic gameplay

Important submission note: For full credit,

- Submit with Phases 1-2 complete by **Monday, March 5** (worth 1 pt).

In the first phase you will complete the implementation that will allow for basic gameplay with the two basic `Ant`s: the `HarvesterAnt` and the `ThrowerAnt`.

Problem 0 (0 pt)

Answer the following questions with your partner after you have read the *entire* `ants.py` file. If you cannot answer these questions, read the file again, consult the core concepts/classes sections above, or ask a question in the Question 0 thread on Piazza.

1. What is the significance of an insect's `armor` attribute? Does this value change? If so, how?
2. What are all of the attributes of the `Ant` class?
3. Is the `armor` attribute of the `Ant` class an instance attribute or class attribute? Why?
4. Is the `damage` attribute of an `Ant` subclass (such as

ThrowerAnt) an instance attribute or class attribute? Why?
5. Which class do both `Ant` and `Bee` inherit from?
6. What do instances of Ant and instances of Bee have in common?
7. How many insects can be in a single Place at any given time (until Phase 6 is complete)?

You can test your understanding by running

```
python3 ok -q 00 -u
```

Problem 1 (1 pt)

First, add food costs and implement harvesters. Currently, there is no cost for deploying any type of `Ant`, and so there is no challenge to the game. You'll notice that `Ant` starts out with a base `food_cost` of zero. Override this value in each of the subclasses listed below with the correct costs.

Class	Food Cost	Armor
`HarvesterAnt`	2	1
`ThrowerAnt`	3	1

Now that deploying `Ant`s cost food, we need to be able gather more food! To fix this issue, implement the `HarvesterAnt` class. A `HarvesterAnt` is a type of `Ant` that adds one food to the `colony.food` total as its `action`.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 01 -u
```

After writing code, test your implementation:

```
python3 ok -q 01
```

Try playing the game again (`python3 gui.py`). Once you have placed a `HarvesterAnt`, you should accumulate food each turn. You can also place `ThrowerAnt`s, but you'll see that they can only attack bees that are in their `Place`, so it'll be a little difficult to win.

Problem 2 (2 pt)

Complete the `Place` constructor by adding code that tracks entrances. Right now, a `Place` keeps track only of its `exit`. We would like a `Place` to keep track of its entrance as well.
A `Place` needs to track only one `entrance`. Tracking entrances will be useful when an `Ant` needs to see what `Bee`s are in front of it in the tunnel.

However, simply passing an entrance to a `Place` constructor will be problematic; we would need to have both the exit and the entrance before creating a `Place`! (It's a chicken or the egg problem.) To get around this problem, we will keep track of entrances in the following way instead. The `Place` constructor should specify that:

- A newly created `Place` always starts with its `entrance` as `None`.
- If the `Place` has an `exit`, then the `exit`'s `entrance` is set to that `Place`.

Hint: Remember that when inside the definition of an `__init__` method, the name `self` is bound to the newly created object.
Hint: Try drawing out two `Place`s next to each other if things get confusing. In the GUI, a place's `entrance`is to its right while the exit is to its left.
Before writing any code, test your understanding of the problem:

```
python3 ok -q 02 -u
```

After writing code, test your implementation:

```
python3 ok -q 02
```

Problem 3 (1 pt)

In order for a `ThrowerAnt` to attack, it must know which bee it should hit. The provided implementation of the `nearest_bee` method in the `ThrowerAnt` class only allows them to hit bees in the same `Place`. Your job is to fix it so that a `ThrowerAnt` will `throw_at` the nearest bee in front of it that is not still in the `Hive`.

The `nearest_bee` method returns a random `Bee` from the nearest place that contains bees. Places are inspected in order by following their `entrance` attributes.

- Start from the current `Place` of the `ThrowerAnt`.
- For each place, return a random bee if there is any, or consider the next place that is stored as the current place's `entrance`.
- If there is no bee to attack, return `None`.

Hint: The `random_or_none` function provided in `ants.py` returns a random element of a sequence or `None` if the sequence is empty.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 03 -u
```

After writing code, test your implementation:

```
python3 ok -q 03
```

After implementing `nearest_bee`, a `ThrowerAnt` should be able to `throw_at` a `Bee` in front of it that is not still in the `Hive`. Make sure that your ants do the right thing! To start a game with ten food (for easy testing):

```
python3 gui.py --food 10
```

Phase 2: Ants Attack

Now that you've implemented basic gameplay with two types of `Ant`s, let's add some flavor to the ways ants can attack bees. In this phase, you'll be implementing several different `Ant`s with different offensive capabilities.

After you implement each `Ant` subclass in this section, you'll need to set its `implemented` attribute to `True` so that that type of ant will show up in the GUI. Feel free to try out the game with each new ant to test the functionality!

With your Phase 2 ants, try `python3 ants_gui.py -d easy` to play against a full swarm of bees in a multi-tunnel layout and try `-d normal`, `-d hard`, or `-d insane` if you want a real challenge! If the bees are too numerous to vanquish, you might need to create some new ants.

Problem 4 (2 pt)

The `ThrowerAnt` is a great offensive unit, but it'd be nice to have a cheaper unit that can throw. Implement two subclasses of `ThrowerAnt` that are less costly but have constraints on the distance they can throw:

- The `LongThrower` can only `throw_at` a `Bee` that is found after following at least 5 `entrance` transitions. It cannot hit `Bee`s that are in the same `Place` as it or the first 4 `Place`s in front of it. If there are two `Bees`, one too close to the `LongThrower` and the other within its range, the `LongThrower` should throw past the closer `Bee`, instead targeting the farther one, which is within its range.
- The `ShortThrower` can only `throw_at` a `Bee` that is found after following at most 3 `entrance` transitions. It can only hit `Bee`s in the same `Place` as it and 3 `Place`s in front of it.

Neither of these specialized throwers can `throw_at` a `Bee` that is exactly 4 `Place`s away. Placing a single one of these (and no other ants) should never win a default game.

Class	Food Cost	Armor
`ShortThrower`	2	1
`LongThrower`	2	1

A good way to approach the implementation to `ShortThrower` and `LongThrower` is to have it inherit the `nearest_bee` method from the base `ThrowerAnt` class. The logic of choosing which bee a thrower ant will attack is essentially the same, except the `ShortThrower` and `LongThrower` ants have maximum and minimum ranges, respectively.

To implement these behaviors, you may need to modify the `nearest_bee` method to reference `min_range` and `max_range` attributes, and only return a bee that is in range.

The original `ThrowerAnt` has no minimum or maximum range, so make sure that its `min_range` and `max_range`attributes should reflect that. Then, implement the subclasses `LongThrower` and `ShortThrower` with appropriately constrained ranges and correct food costs.

Hint: `float('inf')` returns an infinite positive value represented as a float that can be compared with other numbers.

Don't forget to set the `implemented` class attribute of `LongThrower` and `ShortThrower` to `True`.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 04 -u
```

After writing code, test your implementation:

```
python3 ok -q 04
```

Problem 5 (2 pt)

Implement the `FireAnt`. A `FireAnt` has a special `reduce_armor` method: when the `FireAnt`'s armor reaches zero or lower, it will reduce the armor of *all* `Bee`s in the same `Place` as the `FireAnt` by its `damage` attribute (defaults to `3`).

Class	Food Cost	Armor
`FireAnt`	5	1

Hint: Damaging a bee may cause it to be removed from its place. If you iterate over a list, but change the contents of that list at the same time, you may not visit all the elements. As the Python tutorial suggests, "If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy." You can copy a list by calling the `list`constructor or slicing the list from the beginning to the end.

Once you've finished implementing the `FireAnt`, give it a class attribute `implemented` with the value `True`.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 05 -u
```

After writing code, test your implementation:

```
python3 ok -q 05
```

You can also test your program by playing a game or two!

A `FireAnt` should destroy all co-located Bees when it is stung. To start a game with ten food (for easy testing):

```
python3 gui.py --food 10
```

Problem 6 (2 pt)

Implement the `HungryAnt`, which will select a random `Bee` from its `place` and eat it whole. After eating a `Bee`, it must spend 3 turns digesting before eating again.

Class	Food Cost	Armor
`HungryAnt`	4	1

Give `HungryAnt` a `time_to_digest` class attribute that holds the number of turns that it takes a `HungryAnt` to digest (default to 3). Also, give each `HungryAnt` an instance attribute `digesting` that counts the number of turns it has left to digest (default is 0, since it hasn't eaten anything at the beginning).

Implement the `action` method of the `HungryAnt` to check if it's digesting; if so, decrement its `digesting` counter. Otherwise, eat a random `Bee` in its `place` by reducing the `Bee`'s armor to 0 and restart the `digesting` timer.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 06 -u
```

After writing code, test your implementation:

```
python3 ok -q 06
```

Problem 7 (2 pt)

Implement the `NinjaAnt`, which damages all `Bee`s that pass by, but can never be stung.

Class	Food Cost	Armor
`NinjaAnt`	5	1

A `NinjaAnt` does not block the path of a `Bee` that flies by. To implement this behavior, first modify the `Ant` class to include a new class attribute `blocks_path` that is `True` by default. Set the value of `blocks_path` to `False` in the `NinjaAnt` class.

Second, modify the `Bee`'s method `blocked` to return `False` if either there is no `Ant` in the `Bee`'s `place` or if there is an `Ant`, but its `blocks_path` attribute is `False`. Now `Bee`s will just fly past `NinjaAnt`s.

Finally, we want to make the `NinjaAnt` damage all `Bee`s that fly past. Implement the `action` method in `NinjaAnt`to reduce the armor of all `Bee`s in the same `place` as the `NinjaAnt` by its `damage` attribute. Similar to the `FireAnt`, you must iterate over a list of bees that may change.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 07 -u
```

After writing code, test your implementation:

```
python3 ok -q 07
```

For a challenge, try to win a game using only `HarvesterAnt` and `NinjaAnt`.

Phase 3: But They Also Protect

Important submission note: For full credit,

- Submit with Phases 3-4 complete by **Thursday, March 8**.

You will get an extra credit point for submitting the entire project by Wednesday, March 7.

We now have some great offensive troops to help vanquish the bees, but let's make sure we're also keeping our defensive efforts up. In this phase you will implement ants that have special defensive capabiltities such as increased armor and the ability to protect other ants.

Problem 8 (1 pt)

We are going to add some protection to our glorious `AntColony` by implementing the `WallAnt`, which is an ant that does nothing each turn. A `WallAnt` is useful because it has a large `armor` value.

Class	Food Cost	Armor
`WallAnt`	4	4

Unlike with previous ants, we have not provided you with a class header. Implement the `WallAnt` class from scratch. Give it a class attribute `name` with the value `'Wall'` (so that the graphics work) and a class attribute`implemented` with the value `True` (so that you can use it in a game).

Before writing any code, test your understanding of the problem:

```
python3 ok -q 08 -u
```

After writing code, test your implementation:

```
python3 ok -q 08
```

Problem 9 (4 pt)

Right now, our ants are quite frail. We'd like to provide a way to help them last longer against the onslaught of the bees. Enter the `BodyguardAnt`.

Class	Food Cost	Armor
`BodyguardAnt`	4	2

A `BodyguardAnt` differs from a normal ant because it is a `container`; it can contain another ant and protect it, all in one `Place`. When a `Bee` stings the ant in a `Place` where one ant contains another, only the container is damaged. The ant inside the container can still perform its original action. If the container perishes, the contained ant still remains in the place (and can then be damaged).

Each `BodyguardAnt` has an instance attribute `ant` that stores the ant it contains. It initially starts off as `None`, to indicate that no ant is being protected. Implement the `contain_ant` method so that it sets the bodyguard's `ant`instance attribute to the passed in `ant` argument. Also implement the `BodyguardAnt`'s `action` method to perform its `ant`'s action if it is currently containing an ant.

In addition, you will need to make the following modifications throughout your program so that a container and its contained ant can both occupy a place at the same time (a maximum of two ants per place), but only if exactly one is a `container`:

1. Add an `Ant.container` class attribute that indicates whether a subclass of `Ant` is a container. For all `Ant`instances, except

for `BodyguardAnt` instances, `container` should be `False`. The `BodyguardAnt.container`attribute should be `True`.
2. Implement the method `Ant.can_contain` which takes an `other` ant as an argument and returns `True` if:
   - This ant is a container.
   - This ant does not already contain another ant.
   - The other ant is not a container.
3. Modify `Place.add_insect` to allow a container and a non-container ant to occupy the same place according to the following rules:
   - If the `ant` currently occupying a `Place` can contain the `insect` (an `Ant`) passed to `add_insect`, then it does.
   - If the `insect` (an `Ant`) passed to `add_insect` can contain the `ant` currently occupying a `Place`, then it does. Also, set the `Place`'s `ant` to be the container insect.
   - If neither `Ant` can contain the other, raise the same `AssertionError` as before (the one already present in the starter code).

Before writing any code, test your understanding of the problem:

```
python3 ok -q 09 -u
```

After writing code, test your implementation:

```
python3 ok -q 09
```

Problem 10 (1 pt)

The `BodyguardAnt` provides great defense, but they say the best defense is a good offense. The `TankAnt` is a container that protects an ant in its place and also deals 1 damage to all bees in its place each turn.

Class	**Food Cost**	**Armor**

`TankAnt`	6	2

You should not need to modify any code outside of the `TankAnt` class. If you find yourself needing to make changes elsewhere, look for a way to write your code for the previous question such that it applies not just to `BodyguardAnt` and `TankAnt` objects, but to `container` ants in general.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 10 -u
```

After writing code, test your implementation:

```
python3 ok -q 10
```

Phase 4: Water and Might

In the final phase, you're going to add one last kick to the game by introducing a new type of place and new ants that are able to occupy this place. One of these ants is the most important ant of them all: the queen of the colony.

Problem 11 (1 pt)

Let's add water to the colony! Currently there are only two types of places, the `Hive` and a basic `Place`. To make things more interesting, we're going to create a new type of `Place` called `Water`.

Only an ant that is `watersafe` can be deployed to a `Water` place. In order to determine whether an `Insect` is `watersafe`, add a new attribute to the `Insect` class named `watersafe` that is `False` by default. Since bees can fly, make their `watersafe` attribute `True`, overriding the default.

Now, implement the `add_insect` method for `Water`. First call `Place.add_insect` to add the insect, regardless of whether it is watersafe. Then, if the insect is not watersafe, reduce the insect's armor to 0 by invoking`reduce_armor`. *Do not copy and paste code.* Instead, use methods that have already been defined and make use of inheritance to reuse the functionality of the `Place` class.

Before writing any code, test your understanding of the problem:

```
python3 ok -q 11 -u
```

After writing code, test your implementation:

```
python3 ok -q 11
```

Once you've finished this problem, play a game that includes water. To access the `wet_layout` which includes water, add the `--water` option (or `-w` for short) when you start the game.

```
python3 gui.py --water
```

Problem 12 (1 pt)

Currently there are no ants that can be placed on `Water`. Implement the `ScubaThrower`, which is a subclass of `ThrowerAnt` that is more costly and `watersafe`, but otherwise identical to its base class. A `ScubaThrower` should not lose its armor when placed in `Water`.

Class	Food Cost	Armor
`ScubaThrower`	6	1

We have not provided you with a class header. Implement the `ScubaThrower` class from scratch. Give it a class attribute `name` with the value `'Scuba'` (so that the graphics work) and remember to set the class attribute`implemented` with the value `True` (so that you can use it in a game).

Before writing any code, test your understanding of the problem:

```
python3 ok -q 12 -u
```

After writing code, test your implementation:

```
python3 ok -q 12
```

Problem 13 (4 pt)

Finally, implement the `QueenAnt`. The queen is a waterproof `ScubaThrower` that inspires her fellow ants through her bravery. The `QueenAnt` doubles the damage of all the ants behind her each time she performs an action. Once an ant's damage has been doubled, it is *not* doubled again for subsequent turns.

Class	**Food Cost**	**Armor**
`QueenAnt`	7	1

However, with great power comes great responsibility. The `QueenAnt` is governed by three special rules:

1. If the queen ever has its armor reduced to 0, the bees win. The bees also still win if any bee reaches the end of a tunnel. You can call `bees_win()` to signal to the simulator that the game is over.
2. There can be only one true queen. Any queen instantiated beyond the first one is an impostor, and should have its armor reduced to 0 upon taking its first action, without doubling any ant's damage or throwing anything. If an impostor dies, the game should still continue as normal.
3. The true (first) queen cannot be removed. Attempts to remove

the queen should have no effect (but should not cause an error). You will need to modify the `remove_insect` method of `Place` to enforce this condition.

Some hints:

- All instances of the same class share the same class attributes. How can you use this information to tell whether a QueenAnt instance is the true QueenAnt?
- You can find each `Place` in a tunnel behind the `QueenAnt` by starting at the ant's `place.exit` and then repeatedly following its `exit`. To detect whether a `Place` is at the end of a tunnel, check whether its `exit`is `None`.
- To make sure that you don't double the damage of the same ant twice, maintain a list of all the ants that have been doubled.
- Remember that there can be two ants in the same place; if this is the case, make sure to double both their damage attributes.
- You may find the `isinstance` function useful for checking if something is an instance of an object. For example:

```
>>> a = Foo()
>>> isinstance(a, Foo)
True
```

Before writing any code, test your understanding of the problem:

```
python3 ok -q 13 -u
```

After writing code, test your implementation:

```
python3 ok -q 13
```

Extra Credit (2 pt)

During Office Hours and Project Parties, the staff will prioritize helping students with required questions. We will not be offering help with this question unless the queue is empty.

Implement two final thrower ants that do zero damage, but instead produce a temporary "effect" on the `action`method of a `Bee` instance that they `throw_at`. This effect is an alternative action that lasts for a certain number of `.action(colony)` calls, after which the `Bee`'s action reverts to its original behavior.

We will be implementing two new ants that subclass `ThrowerAnt`.

- `SlowThrower` applies a slow effect for 3 turns.
- `StunThrower` applies a stun effect for 1 turn.

Class	Food Cost	Armor
`SlowThrower`	4	1
`StunThrower`	6	1

In order to complete the implementations of these two ants, you will need to set their class attributes appropriately and implement the following three functions:

1. `make_slow` is an effect that takes an `action` method and returns a new `action` method that performs the original action on turns where `colony.time` is even and does nothing on other turns.
2. `make_stun` is an effect that takes an `action` method and returns a new `action` method that does nothing.
3. `apply_effect` takes an `effect` (either `make_slow` or `make_stun`), a `Bee`, and a `duration`. It uses the effect on the `Bee`'s `.action` method to produce a new `action` method, and then arranges to have the new method become the bee's action method for the

next `duration` times that `.action` is called, after which the previous `.action` method is restored.

You can run some provided tests, but they are not exhaustive:

```
python3 ok -q EC -u
python3 ok -q EC
```

Make sure to test your code! Your code should be able to apply multiple effects on a target; each new effect applies to the current (possibly affected) action method of the bee.

Conclusion

You are now done with the project! If you haven't yet, you should try playing the game! There are two GUIs that you can use. The first is a new browser GUI that has fancy graphics and animations. The command to run it is:

```
python3 gui.py [-h] [-d DIFFICULTY] [-w] [--food
FOOD]
```

The second is an older, but tried-and-true interface that we have been using over the past few years. The command to run it is:

```
python3 ants_gui.py [-h] [-d DIFFICULTY] [-w] [--
food FOOD]
```

Acknowledgments: Tom Magrino and Eric Tzeng developed this project with John DeNero. Jessica Wan contributed the original artwork. Joy Jeng and Mark Miyashita invented the queen ant. Many others have contributed to the project as well!

Colin Schoen developed the new browser GUI. The beautiful new artwork was drawn by the efforts of Alana Tran, Andrew Huang, Emilee Chen, Jessie Salas, Jingyi Li, Katherine Xu, Meena Vempaty, Michelle Chang, and Ryan Davis.