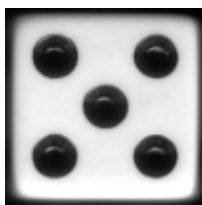


Project 1: The Game of Hog

hog.zip (hog.zip)



*I know! I'll use my
Higher-order functions to
Order higher rolls.*

Introduction

Important submission note: For full credit:

- Submit with Phase 1 complete by **Thursday, July 1** (worth 1 pt).
- Submit with all phases complete by **Wednesday, July 7**.

Although Phase 1 is due only a few days before the rest of the project, you should not put off completing Phase 1. We recommend starting and finishing Phase 1 as soon as possible.

The entire project can be completed with a partner.

You can get 1 bonus point by submitting the entire project by **Tuesday, July 6**.

In this project, you will develop a simulator and multiple strategies for the dice game Hog. You will need to use *control statements* and *higher-order functions* together, as described in Sections 1.2 through 1.6 of Composing Programs (<http://composingprograms.com>).

Rules

In Hog, two players alternate turns trying to be the first to end a turn with at least 100 total points. On each turn, the current player chooses some number of dice to roll, up to 10. That player's score for the turn is the sum of the dice outcomes. However, a player who rolls too many dice risks:

- **Sow Sad.** If any of the dice outcomes is a 1, the current player's score for the turn

is 1.

Examples

- *Example 1:* The current player rolls 7 dice, 5 of which are 1's. They score 1 point for the turn.
- *Example 2:* The current player rolls 4 dice, all of which are 3's. Since Sow Sad did not occur, they score 12 points for the turn.

In a normal game of Hog, those are all the rules. To spice up the game, we'll include some special rules:

- **Piggy Points.** A player who chooses to roll zero dice scores $k + 4$ points, where k is the absolute value of the tens digit minus the ones digit in the opponent's score. If the opponent's score is only one digit, assume the tens digit is 0. You may **not** assume that the score is under 100.

Examples

- *Example 1:* The current player rolls zero dice and the opponent has a score of 14. $|1 - 4| = 3$, so the current player will receive $3 + 4 = 7$ points.
- *Example 2:* The current player rolls zero dice and the opponent has a score of 50. $|5 - 0| = 5$, so the current player will receive $5 + 4 = 9$ points.
- *Example 3:* The current player rolls zero dice and the opponent has a score of 9. $|0 - 9| = 9$, so the current player will receive $9 + 4 = 13$ points.
- *Example 4:* The current player rolls zero dice and the opponent has a score of 156. $|5 - 6| = 1$, so the current player will receive $1 + 4 = 5$ points.
- **More Boar.** After the points for the turn are added to the current player's score, the current player takes an extra turn if the minimum digit of the current player's score is strictly *smaller* than the minimum digit of the opponent's score **and** the maximum digit of the current player's score is strictly *larger* than the maximum digit of the opponent's score. You may **not** assume that the scores are under 100. The More Boar calculation should be done on the current player's score **after** the points from the current turn are added. More Boar can be activated multiple times in a row for the same player (see Example 4).

Examples

- *Example 1:* After the points are added, the current player has a score of 25 and the opponent has a score of 43. Since $2 < 3$, and $5 > 4$, the current player takes an extra turn.
- *Example 2:* After the points are added, the current player has a score of 32 and the opponent has a score of 33. Since the current player's maximum digit (3) is not strictly larger than the opponent's maximum digit ($3 = 3$), the current player does *not* take an extra turn.
- *Example 3:* After the points are added, the current player has a score of 7 and the

opponent has a score of 10. Since the current player's score only has one digit, 7 is both the maximum and minimum digit of the current player's score, and 0 is the minimum digit of the opponent's score. Since $7 \geq 0$, the current player does *not* take an extra turn.

- *Example 4:* After the points are added, the current player has a score of 25 and the opponent has a score of 43. Like in Example 1, the current player takes an extra turn. If the current player then rolls a 1 and now has a score of 26, More Boar activates again and the current player takes yet another extra turn.

Final Product

Our staff solution to the project can be played at hog.cs61a.org (<https://hog.cs61a.org>) -- try it out! When you finish the project, you'll have implemented a significant part of this game yourself.

Download starter files

To get started, download all of the project code as a zip archive (hog.zip). Below is a list of all the files you will see in the archive. However, you only have to make changes to `hog.py`.

- `hog.py` : A starter implementation of Hog
- `dice.py` : Functions for rolling dice
- `hog_gui.py` : A graphical user interface (GUI) for Hog
- `ucb.py` : Utility functions for CS 61A
- `ok` : CS 61A autograder
- `tests` : A directory of tests used by `ok`
- `gui_files` : A directory of various things used by the web GUI
- `calc.py` : A file you can use to approximately test your final strategy

You may notice some files other than the ones listed above too -- those are needed for making the autograder and portions of the GUI work. Please do not modify any files other than `hog.py`.

Logistics

The project is worth 24 points. 23 points are assigned for correctness and 1 point for submitting Part I by the checkpoint date.

You will turn in the following files:

- `hog.py`

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 (<http://ok.cs61a.org>).

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.

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. **If you forget to submit, your last backup will be automatically converted to a submission.**

If you do not want us to record a backup of your work or information about your progress, you can run

```
python3 ok --local
```

With this option, no information will be sent to our course servers. If you want to test your code interactively, you can run

```
python3 ok -q [question number] -i
```

with the appropriate question number (e.g. `01`) inserted. This will run the tests for that question until the first one you failed, then give you a chance to test the functions you wrote interactively.

You can also use the debugging print feature in OK by writing

```
print("DEBUG:", x)
```

which will produce an output in your terminal without causing OK tests to fail with extra output.

Graphical User Interface

A **graphical user interface** (GUI, for short) is provided for you. At the moment, it doesn't work because you haven't implemented the game logic. Once you complete the `play` function, you will be able to play a fully interactive version of Hog!

Once you've done that, you can run the GUI from your terminal:

```
python3 hog_gui.py
```

The GUI is an open-source project hosted on Github (<https://github.com/Cal-CS-61A-Staff/hog-gui>).

Phase 1: Simulator

In the first phase, you will develop a simulator for the game of Hog.

Getting Started Video

Phase 1 Getting Started Videos

This video provides some helpful direction for tackling the problems on this phase of Hog.

Problem 0 (0 pt)

The `dice.py` file represents dice using non-pure zero-argument functions. These functions are non-pure because they may have different return values each time they are called. The documentation of `dice.py` describes the two different types of dice used in the project:

- A **fair** dice produces each possible outcome with equal probability. Two fair dice are already defined, `four_sided` and `six_sided`, and are generated by the `make_fair_dice` function.
- A **test** dice is deterministic: it always cycles through a fixed sequence of values that are passed as arguments. Test dice are generated by the `make_test_dice` function.

Before writing any code, read over the `dice.py` file and check your understanding by unlocking the following tests.

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

This should display a prompt that looks like this:

```
=====
Assignment: Project 1: Hog
Ok, version v1.5.2
=====

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)

>>> test_dice = make_test_dice(4, 1, 2)
>>> test_dice()
?
```

You should type in what you expect the output to be. To do so, you need to first figure out what `test_dice` will do, based on the description above.

You can exit the unlocker by typing `exit()`. **Typing Ctrl-C on Windows to exit out of the unlocker has been known to cause problems, so avoid doing so.**

In general, for each of the unlocking tests, you might find it helpful to read through the provided skeleton for that problem before attempting the unlocking test.

Problem 1 (2 pt)

Implement the `roll_dice` function in `hog.py`. It takes two arguments: a positive integer called `num_rolls` giving the number of dice to roll and a `dice` function. It returns the number of points scored by rolling the dice that number of times in a turn: either the sum of the outcomes or 1 (*Sow Sad*).

The Sow Sad rule:

- **Sow Sad.** If any of the dice outcomes is a 1, the current player's score for the turn is 1.

Examples

- *Example 1:* The current player rolls 7 dice, 5 of which are 1's. They score 1 point for the turn.
- *Example 2:* The current player rolls 4 dice, all of which are 3's. Since Sow Sad did not occur, they score 12 points for the turn.

To obtain a single outcome of a dice roll, call `dice()`. You should call `dice()` exactly `num_rolls` times in the body of `roll_dice`. **Remember to call `dice()` exactly `num_rolls` times even if **Sow Sad** happens in the middle of rolling.** In this way, you correctly simulate rolling all the dice together.

Understand the problem:

Before writing any code, unlock the tests to verify your understanding of the question.

Note: you will not be able to test your code using OK until you unlock the test cases for the corresponding question.

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

Write code and check your work:

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 01
```

Debugging Tips

If the tests don't pass, it's time to debug. You can observe the behavior of your function using Python directly. First, start the Python interpreter and load the `hog.py` file.

```
python3 -i hog.py
```

Then, you can call your `roll_dice` function on any number of dice you want. The `roll_dice` function has a default argument value (<http://composingprograms.com/pages/14-designing-functions.html#default-argument-values>) for `dice` that is a random six-sided dice function. Therefore, the following call to `roll_dice` simulates rolling four fair six-sided dice.

```
>>> roll_dice(4)
```

You will find that the previous expression may have a different result each time you call it, since it is simulating random dice rolls. You can also use test dice that fix the outcomes of the dice in advance. For example, rolling twice when you know that the dice will come up 3 and 4 should give a total outcome of 7.

```
>>> fixed_dice = make_test_dice(3, 4)
>>> roll_dice(2, fixed_dice)
7
```

On most systems, you can evaluate the same expression again by pressing the up arrow, then pressing enter or return. To evaluate earlier commands, press the up arrow repeatedly.

If you find a problem, you need to change your `hog.py` file, save it, quit Python, start Python again, and then start evaluating expressions. Pressing the up arrow should give you access to your previous expressions, even after restarting Python.

Continue debugging your code and running the `ok` tests until they all pass. You should follow this same procedure of understanding the problem, implementing a solution, testing, and debugging for all the problems in this project.

One more debugging tip: to start the interactive interpreter automatically upon failing an `ok` test, use `-i`. For example, `python3 ok -q 01 -i` will run the tests for question 1, then start an interactive interpreter with `hog.py` loaded if a test fails.

Problem 2 (1 pt)

Implement `piggy_points`, which takes the opponent's current `score` and returns the number of points scored by rolling 0 dice.

The Piggy Points rule:

- **Piggy Points.** A player who chooses to roll zero dice scores $k + 4$ points, where k is the absolute value of the tens digit minus the ones digit in the opponent's score. If the opponent's score is only one digit, assume the tens digit is 0. You may **not** assume that the score is under 100.

Examples

- *Example 1:* The current player rolls zero dice and the opponent has a score of 14. $|1 - 4| = 3$, so the current player will receive $3 + 4 = 7$ points.
- *Example 2:* The current player rolls zero dice and the opponent has a score of 50. $|5 - 0| = 5$, so the current player will receive $5 + 4 = 9$ points.
- *Example 3:* The current player rolls zero dice and the opponent has a score of 9. $|0 - 9| = 9$, so the current player will receive $9 + 4 = 13$ points.
- *Example 4:* The current player rolls zero dice and the opponent has a score of 156. $|5 - 6| = 1$, so the current player will receive $1 + 4 = 5$ points.

You are allowed to implement this function in any way you want **as long as you do not use for loops or square brackets [] in your implementation**. You don't have to follow this recommended method or use the provided starter code.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 02
```

You can also test `piggy_points` interactively by entering `python3 -i hog.py` in the terminal and then calling `piggy_points` with various inputs.

Problem 3 (2 pt)

Implement the `take_turn` function, which returns the number of points scored for a turn by rolling the given `dice` `num_rolls` times.


Your implementation of `take_turn` should call both `roll_dice` and `piggy_points` when possible.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 03
```

 Pair programming? ([/~cs61a/su21/articles/pair-programming](https://inst.eecs.berkeley.edu/~cs61a/su21/articles/pair-programming)) Remember to alternate between driver and navigator roles! The driver controls the keyboard; the navigator watches, asks questions, and suggests ideas.

Problem 4 (2 pt)

Implement `more_boar`, which takes the current player and opponent scores and returns whether the current player will take another turn due to More Boar.

Hint: You might find the Python builtin `max` and `min` functions useful.

The More Boar rule:

- **More Boar.** After the points for the turn are added to the current player's score, the current player takes an extra turn if the minimum digit of the current player's score is strictly *smaller* than the minimum digit of the opponent's score **and** the maximum digit of the current player's score is strictly *larger* than the maximum digit of the opponent's score. You may **not** assume that the scores are under 100. The More Boar calculation should be done on the current player's score **after** the points from the current turn are added. More Boar can be activated multiple times in a row for the same player (see Example 4).

Examples

- *Example 1:* After the points are added, the current player has a score of 25 and the opponent has a score of 43. Since $2 < 3$, and $5 > 4$, the current player takes an extra turn.
- *Example 2:* After the points are added, the current player has a score of 32 and the opponent has a score of 33. Since the current player's maximum digit (3) is not strictly larger than the opponent's maximum digit ($3 = 3$), the current player does *not* take an extra turn.
- *Example 3:* After the points are added, the current player has a score of 7 and the opponent has a score of 10. Since the current player's score only has one digit, 7 is both the maximum and minimum digit of the current player's score, and 0 is the minimum digit of the opponent's score. Since $7 \geq 0$, the current player does *not* take an extra turn.
- *Example 4:* After the points are added, the current player has a score of 25 and the opponent has a score of 43. Like in Example 1, the current player takes an extra turn. If the current player then rolls a 1 and now has a score of 26, More Boar activates again and the current player takes yet another extra turn.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 04
```

Problem 5 (4 pt)

Implement the `play` function, which simulates a full game of Hog. Players take turns

rolling dice until one of the players reaches the `goal` score.

A turn is defined as one roll of the dice, as the same player can take multiple turns in a row.

To determine how many dice are rolled each turn, each player uses their respective strategy (Player 0 uses `strategy0` and Player 1 uses `strategy1`). A *strategy* is a function that, given a player's score and their opponent's score, returns the number of dice that the current player will roll in the turn. Don't worry about implementing strategies yet; you'll do that in Phase 3.

When the game ends, `play` returns the final total scores of both players, with Player 0's score first and Player 1's score second.

Note: Each strategy function should be called only once per turn. This means please only call `strategy0` when it is Player 0's turn and only call `strategy1` when it is Player 1's turn. Otherwise, the GUI and some ok tests will get confused.

Note: When one player achieves the goal score, the game ends immediately and the other player doesn't get to play another turn.

Hints:

- You should call the functions you have implemented already.
- Call `take_turn` with four arguments (don't forget to pass in the `goal`). Only call `take_turn` once per turn.
- Call `more_boar` to determine if the current player will take another turn due to More Boar.
- You can get the number of the next player (either 0 or 1) by calling the provided function `next_player`.
- You can ignore the `say` argument to the `play` function for now. You will use it in Phase 2 of the project.
- For the unlocking tests, `hog.always_roll` refers to the `always_roll` function defined in `hog.py`.

Rules Clarification: A player can take more than two consecutive turns. For example, if the score after their first turn is 29 vs 55, they go again due to More Boar. If they score 9 points and now the score is 38 vs 55, they go a third time again due to More Boar. If they score 10 points and now the score is 48 vs 55, they go a fourth time in a row.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 05
```

Once you are finished, you will be able to play a graphical version of the game. We have provided a file called `hog_gui.py` that you can run from the terminal:

```
python3 hog_gui.py
```

The GUI relies on your implementation, so if you have any bugs in your code, they will be reflected in the GUI. This means you can also use the GUI as a debugging tool; however, it's better to run the tests first.


Make sure to submit your work so far before the checkpoint deadline:

```
python3 ok --submit
```

Check to make sure that you did all the problems in Phase 1:

```
python3 ok --score
```

Congratulations! You have finished Phase 1 of this project!

 Pair programming? (</~cs61a/su21/articles/pair-programming>) This is a good time to switch roles! Switching roles makes sure that you both benefit from the learning experience of being in each role.

Phase 2: Commentary

In the second phase, you will implement commentary functions that print remarks about the game after each turn, such as, "Player 1 has reached a new maximum point gain. 22 point(s)!"

A commentary function takes two arguments, Player 0's current score and Player 1's current score. It can print out commentary based on either or both current scores and any other information in its parent environment. Since commentary can differ from turn to turn depending on the current point situation in the game, a commentary function always returns another commentary function to be called on the next turn. The only side effect of a commentary function should be to print.

Commentary examples

The function `say_scores` in `hog.py` is an example of a commentary function that simply announces both players' scores. Note that `say_scores` returns itself, meaning that the same commentary function will be called each turn.

```
def say_scores(score0, score1):
    """A commentary function that announces the score for each player."""
    print("Player 0 now has", score0, "and Player 1 now has", score1)
    return say_scores
```

The function `announce_lead_changes` is an example of a higher-order function that returns a commentary function that tracks lead changes. A different commentary function will be called each turn.

```
def announce_lead_changes(last_leader=None):
    """Return a commentary function that announces lead changes.

    >>> f0 = announce_lead_changes()
    >>> f1 = f0(5, 0)
    Player 0 takes the lead by 5
    >>> f2 = f1(5, 12)
    Player 1 takes the lead by 7
    >>> f3 = f2(8, 12)
    >>> f4 = f3(8, 13)
    >>> f5 = f4(15, 13)
    Player 0 takes the lead by 2
    """
    def say(score0, score1):
        if score0 > score1:
            leader = 0
        elif score1 > score0:
            leader = 1
        else:
            leader = None
        if leader != None and leader != last_leader:
            print('Player', leader, 'takes the lead by', abs(score0 - score1))
        return announce_lead_changes(leader)
    return say
```

You should also understand the function `both`, which takes two commentary functions (`f` and `g`) and returns a *new* commentary function. This returned commentary function returns *another* commentary function which calls the functions returned by calling `f` and `g`, in that order.

```
def both(f, g):
    """Return a commentary function that says what f says, then what g says.

    >>> h0 = both(say_scores, announce_lead_changes())
    >>> h1 = h0(10, 0)
    Player 0 now has 10 and Player 1 now has 0
    Player 0 takes the lead by 10
    >>> h2 = h1(10, 8)
    Player 0 now has 10 and Player 1 now has 8
    >>> h3 = h2(10, 17)
    Player 0 now has 10 and Player 1 now has 17
    Player 1 takes the lead by 7
    """
    def say(score0, score1):
        return both(f(score0, score1), g(score0, score1))
    return say
```

Problem 6 (2 pt)

Getting Started Video

Update your `play` function so that a commentary function is called at the end of each turn. The return value of calling a commentary function gives you the commentary function to call on the next turn.

For example, `say(score0, score1)` should be called at the end of the first turn. Its return value (another commentary function) should be called at the end of the second turn. Each consecutive turn, call the function that was returned by the call to the previous turn's commentary function.

Hint: For the unlocking tests for this problem, remember that when calling `print` with multiple arguments, Python will put a space between each of the arguments. For example,

```
>>> print(9, 12)
9 12
```

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your

correctness with:

```
python3 ok -q 06
```

Problem 7 (3 pt)

Getting Started Video

Implement the `announce_highest` function, which is a higher-order function that returns a commentary function. This commentary function announces whenever a particular player gains more points in a turn than ever before. For example, `announce_highest(1)` ignores Player 0 entirely and just prints information about Player 1. (So does its return value; another commentary function about only Player 1.)

To compute the gain, it must compare the score from last turn (`last_score`) to the score from this turn for the player of interest (designated by the `who` argument). This function must also keep track of the highest gain for the player so far, which is stored as `running_high`.

The way in which `announce_highest` announces is very specific, and your implementation should match the doctests provided. Don't worry about singular versus plural when announcing point gains; you should simply use "point(s)" for both cases.

Hint: The `announce_lead_changes` function provided to you is an example of how to keep track of information using commentary functions. If you are stuck, first make sure you understand how `announce_lead_changes` works.

Hint. If you're getting a `local variable [var] reference before assignment error`:

This happens because in Python, you aren't normally allowed to modify variables defined in parent frames. Instead of reassigning `[var]`, the interpreter thinks you're trying to define a new variable within the current frame. We'll learn about how to work around this in a future lecture, but it is not required for this problem.

To fix this, you have two options:

- 1) Rather than reassigning `[var]` to its new value, create a new variable to hold that new value. Use that new variable in future calculations.
- 2) For this problem specifically, avoid this issue entirely by not using assignment statements at all. Instead, pass new values in as arguments to a call to `announce_highest`.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 07
```

When you are done, you will see commentary in the GUI:

```
python3 hog_gui.py
```

The commentary in the GUI is generated by passing the following function as the `say` argument to `play`.

```
both(announce_highest(0), both(announce_highest(1), announce_lead_changes()))
```

Great work! You just finished Phase 2 of the project!



Pair programming? ([/~cs61a/su21/articles/pair-programming](https://inst.eecs.berkeley.edu/~cs61a/su21/articles/pair-programming)) Celebrate, take a break, and switch roles!

Phase 3: Strategies

In the third phase, you will experiment with ways to improve upon the basic strategy of always rolling a fixed number of dice. First, you need to develop some tools to evaluate strategies.

Problem 8 (2 pt)

Getting Started Video

Implement the `make_averaged` function, which is a higher-order function that takes a function `original_function` as an argument. It returns another function that takes the same number of arguments as `original_function` (the function originally passed into `make_averaged`). This returned function differs from the input function in that it returns the average value of repeatedly calling `original_function` on the same arguments. This function should call `original_function` a total of `trials_count` times and return the average of the results.

To implement this function, you need a new piece of Python syntax! You must write a

function that accepts an arbitrary number of arguments, then calls another function using exactly those arguments. Here's how it works.

Instead of listing formal parameters for a function, you can write `*args`. To call another function using exactly those arguments, you call it again with `*args`. For example:

```
>>> def printed(f):
...     def print_and_return(*args):
...         result = f(*args)
...         print('Result:', result)
...         return result
...     return print_and_return
>>> printed_pow = printed(pow)
>>> printed_pow(2, 8)
Result: 256
256
>>> printed_abs = printed(abs)
>>> printed_abs(-10)
Result: 10
10
```

Read the docstring for `make_averaged` carefully to understand how it is meant to work. Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 08
```

Problem 9 (2 pt)

Getting Started Video

Implement the `max_scoring_num_rolls` function, which runs an experiment to determine the number of rolls (from 1 to 10) that gives the maximum average score for a turn. Your implementation should use `make_averaged` and `roll_dice`.

If two numbers of rolls are tied for the maximum average score, return the lower number. For example, if both 3 and 6 achieve a maximum average score, return 3.

You might find it useful to read the doctest and the example shown in the doctest for

this problem before doing the unlocking test.

IMPORTANT NOTE: In order to pass all of our tests, please make sure that you are testing dice rolls **starting from 1 going up to 10**, rather than starting from 10 to 1.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 09
```

Running experiments:


To run this experiment on randomized dice, call `run_experiments` using the `-r` option:

```
python3 hog.py -r
```

For the remainder of this project, you can change the implementation of `run_experiments` as you wish. The function includes calls to `average_win_rate` for evaluating various Hog strategies, but most of the calls are currently commented out. You can un-comment the calls to try out strategies, like to compare the win rate for `always_roll(8)` to the win rate for `always_roll(6)`.

Some of the experiments may take up to a minute to run. You can always reduce the number of trials in your call to `make_averaged` to speed up experiments.

Running experiments won't affect your score on the project.

 Pair programming? (</~cs61a/su21/articles/pair-programming>) We suggest switching roles now, if you haven't recently. Almost done!

Problem 10 (1 pt)

Getting Started Video

A strategy can try to take advantage of the *Piggy Points* rule by rolling 0 when it is most beneficial to do so. **Implement `piggypoints_strategy`, which returns 0 whenever rolling 0 would give at least `cutoff` points and returns `num_rolls` otherwise.**

Hint: You can use the function `piggy_points` you defined in Problem 2

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 10
```

Once you have implemented this strategy, **change `run_experiments` to evaluate your new strategy against the baseline.** Is it better than just rolling 4?

Problem 11 (2 pt)

Getting Started Video

A strategy can also take advantage of the *More Boar* rules. The more boar strategy always rolls 0 if doing so triggers another turn. In other cases, it rolls 0 if rolling 0 would give **at least** `cutoff` points. Otherwise, the strategy rolls `num_rolls`.

Hint: You can use the function `piggypoints_strategy` you defined in Problem 10

Hint: Remember that the `more_boar` check should be done after the points from `piggy_points` have been added to the score.

Before writing any code, unlock the tests to verify your understanding of the question.

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

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 11
```

Once you have implemented this strategy, update `run_experiments` to evaluate your new strategy against the baseline. You should find that it gives a significant edge over `always_roll(6)`.

Optional: Problem 12 (0 pt)

Implement `final_strategy`, which combines these ideas and any other ideas you have to achieve a high win rate against the `always_roll(6)` strategy. Some suggestions:

- `more_boar_strategy` is a good default strategy to start with.
- If you know the goal score (by default it is 100), there's no point in scoring more than the goal. Check whether you can win by rolling 0, 1 or 2 dice. If you are in the lead, you might take fewer risks.
- Try to force another turn.
- Choose the `num_rolls` and `cutoff` arguments carefully.
- Take the action that is most likely to win the game.

You can check that your final strategy is valid by running Ok.

```
python3 ok -q 12
```

You will also eventually be able to check your exact final win rate by running

```
python3 calc.py
```

This should pop up a window asking for you to confirm your identity, and then it will print out a win rate for your final strategy.

You can also play against your final strategy with the graphical user interface:

```
python3 hog_gui.py
```

The GUI will alternate which player is controlled by you.

Project submission

At this point, run the entire autograder to see if there are any tests that don't pass:

```
python3 ok
```

You can also check your score on each part of the project:

```
python3 ok --score
```

Once you are satisfied, submit to Ok to complete the project.

```
python3 ok --submit
```

If you have a partner, make sure to add them to the submission on okpy.org.

Congratulations, you have reached the end of your first CS 61A project! If you haven't already, relax and enjoy a few games of Hog with a friend.