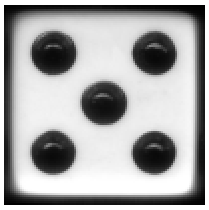
# Project 1: The Game of Hog



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

## Introduction

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).

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.

To spice up the game, we will play with some special rules:

- **Pig Out**. If any of the dice outcomes is a 1, the current player's score for the turn is 0.
- **Piggy Back**. When the current player scores 0, the opposing player receives points equal to the number of dice rolled that turn.
    - *Example*: If the current player rolls 3 dice that come up 1, 5, and 1, then the current player scores 0 and the opponent scores 3.
- **Free Bacon**. A player who chooses to roll zero dice scores one more than the largest digit in the opponent's total score.
    - *Example 1*: If the opponent has 42 points, the current player gains 1 + max(4, 2) = 5 points by rolling zero dice.
    - *Example 2*: If the opponent has has 48 points, the current player gains 1 + max(4, 8) = 9 points by rolling zero dice.
    - *Example 3*: If the opponent has has 7 points, the current player gains 1 + max(0, 7) = 8 points by rolling zero dice.
- **Hog Wild**. If the sum of both players' total scores is a multiple of seven (e.g., 14, 21, 35), then the current player rolls four-sided dice instead of the usual six-sided dice.
- **Hogtimus Prime**. If a player's score for the turn is a prime number, then the turn score is increased to the next largest prime number. For example, if the dice outcomes sum to 19, the current player

scores 23 points for the turn. This boost only applies to the current player. *Note:* 1 is not a prime number!
- **Swine Swap**. After the turn score is added, if the last two digits of each player's score are the reverse of each other, the players swap total scores.
  - *Example 1*: The current player has a total score of 13 and the opponent has 91. The current player rolls two dice that total 6. The last two digits of the current player's new total score (19) are the reverse of the opponent's score (91). These scores are swapped! The current player now has 91 points and the opponent has 19. The turn ends.
  - *Example 2*: The current player has 66 and the opponent has 8. The current player rolls four dice that total 14, leaving the current player with 80. The reverse of 80 is 08, the opponent's score. After the swap, the current player has 8 and the opponent 80. The turn ends.
  - *Example 3*: Both players have 90. The current player rolls 7 dice that total 17, a prime that is boosted to 19 points for the turn. The current player has 109 and the opponent has 90. The last two digits 09 and 90 are the reverse of each other, so the scores are swapped. The opponent ends the turn with 109 and wins the game.

## Download starter files

To get started, download all of the project code as a zip archive (hog.zip). 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 for Hog
- `ucb.py` : Utility functions for CS 61A
- `ok` : CS 61A autograder
- `tests` : A directory of tests used by `ok`
- `images` : A directory of images used by `hog_gui.py`

## Logistics

This is a 1-week 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. The project is worth 20 points. 18 points are assigned for correctness, and 2 points for the overall composition (../../articles/composition.html) of your program.

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).

## Testing

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.

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 is a catch. At first, 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: The Game of Hog
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.

If you are trying to debug a test failure, you can launch an interactive session after the test is run with:

```
python3 ok -q 05 -i
```

This will run the tests and launch an interactive session if a test does not pass.

```
=====================================================================
Assignment: Project 1: Hog
OK, version ....
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Running tests

---------------------------------------------------------------------
Question ... > Suite ... > Case ...

>>> the_test()
"expected value"

# Error: expected
#     "expected value"
# but got
#     None

# Interactive console. Type exit() to quit
>>>
```

The `tests` folder is used to store autograder tests, so make sure **not to 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 (hog.zip) and copy it over, but you will need to start unlocking from scratch.

## 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!

In order to render the graphics, make sure you have Tkinter, Python's main graphics library, installed on your computer. Once you've done that, you can run the GUI from your terminal:

```
python3 hog_gui.py
```

Once you complete the project, you can play against the final strategy that you've created!

```
python3 hog_gui.py -f
```

## Phase 1: Simulator

In the first phase, you will develop a simulator for the game 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:

- Dice can be fair, meaning that they produce each possible outcome with equal probability. Examples: `four_sided`, `six_sided`.
- For testing functions that use dice, deterministic test dice always cycle through a fixed sequence of values that are passed as arguments to the `make_test_dice` function.

Before we start writing any code, let's understand the `make_test_dice` function by unlocking its 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.

**Note:** you can exit the unlocker by typing `exit()` (without quotes). **Typing Ctrl-C on Windows to exit out of the unlocker has been known to cause problems, so avoid doing so.**

## Problem 1 (2 pt)

Implement the `roll_dice` function in `hog.py`. It takes two arguments: the number of dice to roll, `num_rolls`, and a `dice` function. It returns the number of points scored by rolling that number of dice **simultaneously**: either the sum of the outcomes or 0 (pig out).

To obtain a single outcome of a dice roll, call `dice()`. You must call the `dice` function *exactly* the number of times specified by the first argument (even if a 1 is rolled) since we are rolling all dice simultaneously in the game (otherwise tests and the GUI will fail).

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

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

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

```
python3 ok -q 01
```

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. The tests use fixed dice.

## Problem 2 (1 pt)

Implement the `take_turn` function, which returns the number of points scored for a turn by the current player. You will need to implement the *Free bacon* rule. You can assume that `opponent_score` is less than 100. For a score less than 10, assume that the first of two digits is 0. Your implementation should call `roll_dice`.

You should also implement the special *Hogtimus Prime* rule here. Don't forget that it applies to both regular turns and Free Bacon turns! To implement *Hogtimus Prime*, write your own prime functions above the `take_turn` function. One way to do so is to write two functions, `is_prime` and `next_prime`. There are no tests for `is_prime` and `next_prime`, but you can test them on your own using doctests that you create. Remember, 1 isn't prime!

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
```

## Problem 3 (1 pt)

Implement the `select_dice` function, which helps enforce the *Hog wild* special rule. This function takes two arguments: the scores for the current and opposing players. It returns either `four_sided` or `six_sided` dice that will be used during the turn.

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
```

## Problem 4 (1 pt)

To help you implement the *Swine Swap* special rule, write a function called `is_swap` that checks to see if the last two digits of the players' scores are swapped.

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 (3 pt)

Implement the `play` function, which simulates a full game of Hog. Players alternate turns, each using their respective strategy function (Player 0 uses strategy0, etc.), until one of the players reaches the `goal` score. 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.

Here are some hints:

- You should use the functions you have already written! You will need to call `take_turn` with all three arguments.
- Enforce all the remaining special rules: *Piggy Back* (check the result of `take_turn`), *Hog wild* (call `select_dice`), and *Swine Swap* (call `is_swap`)
- You can get the number of the other player (either 0 or 1) by calling the provided function `other`.
- A *strategy* is a function that, given a player's score and their opponent's score, returns how many dice the player wants to roll. A strategy function (such as `strategy0` and `strategy1`) takes two arguments: scores for the current player and opposing player. A strategy function returns the number of dice that the current player wants to roll in the turn. Don't worry about details of implementing strategies yet. You will develop them in Phase 2.

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
```

**Note**: the last test for Question 5 is a *fuzz test*, which checks your `play` function works for any arbitrary inputs. Failing this test means something is wrong, but you should look at other tests to see where the problem might be.

*Hint*: If you fail the fuzz test, check that you're only calling `take_turn` once per turn!

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
```

If you don't already have Tkinter (Python's graphics library) installed, you'll need to install it first before you can run the GUI.

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.

Congratulations! You have finished Phase 1 of this project!

## Phase 2: Strategies

In the second 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 6 (2 pt)

Implement the `make_averaged` function, which is a higher-order function that takes a function `fn` as an argument. It returns another function that takes the same number of arguments as `fn` (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 `fn` on the same arguments. This function should call `fn` a total of `num_samples` 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, we write `*args` . To call another function using exactly those arguments, we call it again with `*args` . For example,

```
>>> def printed(fn):
...     def print_and_return(*args):
...         result = fn(*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 06 -u
```

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

```
python3 ok -q 06
```

## Problem 7 (2 pt)

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` .

**Note:** 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.

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
```

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

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

**Running experiments** For the remainder of this project, you can change the implementation of `run_experiments` as you wish. By calling `average_win_rate` , you can evaluate various Hog strategies. For example, change the first `if False:` to `if True:` in order to evaluate `always_roll(8)` against the baseline strategy of `always_roll(5)` . You should find that it loses more often than it wins, giving a win rate below 0.5.

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

## Problem 8 (1 pt)

A strategy can take advantage of the *Free bacon* rule by rolling 0 when it is most beneficial to do so. Implement `bacon_strategy` , which returns 0 whenever rolling 0 would give **at least** `margin` points and returns `num_rolls` otherwise.

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
```

Once you have implemented this strategy, change `run_experiments` to evaluate your new strategy against the baseline. You should find that it wins more than half of the time.

## Problem 9 (2 pt)

A strategy can also take advantage of the *Swine Swap* rule. The `swap_strategy` rolls 0 if it would cause a beneficial swap and `num_rolls` otherwise. If a swap would result in the scores not changing at all, the strategy should roll `num_rolls` .

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
```

Once you have implemented this strategy, update `run_experiments` to evaluate your new strategy against the baseline. You should find that it gives a slight edge over `always_roll(5)` .

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

```
python3 ok
```

## Problem 10 (3 pt)

Implement `final_strategy` , which combines these ideas and any other ideas you have to achieve a win rate of at least 0.77 (for full credit) against the baseline `always_roll(5)` strategy. Partial credit is also given if you are close. Some ideas:

- Combine the ideas of `swap_strategy` and `bacon_strategy` .
- Choose the `num_rolls` and `margin` arguments carefully.
- Don't swap scores when you're winning.
- There's no point in scoring more than 100. Check for chances to win.

You can check your final strategy win rate by running OK.

```
python3 ok -q 10
```

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

```
python3 hog_gui.py -f
```

The GUI will alternate which player is controlled by you.

Congratulations, you have reached the end of your first CS 61A project!