Project 1: The Game of Hog

Download Project Materials

To get started, download project materials `project1.zip` from our QQ group if you don't have one. Below is a list of all the files you will see in the `project1.zip`. However, you only have to make changes to `hog/hog.py` in this project.

```
project1
  |-hog
  |   |-gui_files   # A directory of various things used by the web gui.
  |   |-dice.py     # Functions for rolling dice
  |   |-hog_gui.py  # A graphical user interface for Hog
  |   |-hog.py      # A starter implementation of Hog
  |   `-ucb.py      # Utility functions from CS 61A
  `-project1.pdf    # Instructions for this project you must read
```

Introduction

> **Important submission notes**: This project has three phases. You have ten days for Phase 1 and five more days for Phase 2 and 3. It doesn't mean that you are restricted to complete Phase 2 and 3 in just five days. That is, you have 15 days in total for this project but the Phase 1 should be finished in the first 10 days. We recommend starting and finishing Phase 1 as soon as possible to give yourself adequate time to complete Phases 2 and 3, which are can be more time consuming. Check the exact deadline on our OJ website.
>
> After completing any problems required in each phase, you need to submit your answer to `Contest 'hog: phase 1'` and `Contest 'hog: phase 2&3'` correspondingly on our OJ website to get your answer scored. We recommend that you submit **after you finish each problem** so that you can find bugs as soon as possible.

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.

Rule

> It's okay to skim the rules below. We will reproduce them on need.

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:

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

- *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 Pig Out 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:

- **Free Bacon**. A player who chooses to roll zero dice scores points equal to one more than the absolute alternating difference of the digits of the opponent's score cubed.
  - *Example 1*: The opponent has 4 points, and the current player chooses to roll zero dice. `4*4*4 = 64`, so the current player gets `1 + |6 - 4| = 3` points.
  - *Example 2*: The opponent has 1 points, and the current player chooses to roll zero dice. `1*1*1 = 1`, so the current player gets `1 + |1| = 2` points.
  - *Example 3*: The opponent has 20 points, and the current player chooses to roll zero dice. `20*20*20 = 8000`, so the current player gets `1 + |8 - 0 + 0 - 0| = 9` points.
  - *Example 4*: The opponent has 45 points, and the current player chooses to roll zero dice. `45*45*45 = 91125`, so the current player gets `1 + |9 - 1 + 1 - 2 + 5| = 13` points.
- **Feral Hogs**. If the number of dice you roll is exactly 2 away (absolute difference) from the number of points you scored on the previous turn, you get 3 extra points for the turn. Treat the turn before the first turn as scoring 0 points. Do **not** take into account any previous feral hog bonuses or swine swap (next rule) when calculating the number of points scored the previous turn.
  - *Example 1*:
    - Both players start out at 0. (0, 0)
    - Player 0 rolls 3 dice and gets 7 points. (7, 0)
    - Player 1 rolls 1 dice and gets 4 points. (7, 4)
    - Player 0 rolls 5 dice and gets 10 points. 5 is 2 away from 7, so player 0 gets a bonus of 3. (20, 4)
    - Player 1 rolls 2 dice and gets 8 points. 2 is 2 away from 4, so player 1 gets a bonus of 3. (20, 15)
    - Player 0 rolls 8 dice and gets 20 points. 8 is 2 away from 10, so player 0 gets a bonus of 3. (43, 15)
    - Player 1 rolls 6 dice and gets 1 point. 6 is 2 away from 8, so player 1 gets a bonus of 3. (43, 19)
  - *Example 2*:
    - Both players start out at 0. (0, 0)
    - Player 0 rolls 2 dice and gets 3 points. 2 is 2 away from 0, so player 0 gets a bonus of 3. (6, 0)
- **Swine Swap**. Define the *excitement* of the game to be three to the power of the sum of the players' scores. After points for the turn are added to the current player's score, if the excitement's first digit and last digit are the same, the scores should be swapped.
  - *Example 1*: At the end of a turn, the players have scores of 2 and 4; `3**(2 + 4) = 729`, and `7 != 9`, the scores are not swapped.

- *Example 2*: At the end of a turn, the players have scores of 11 and 1; `3**(11 + 1) = 531441`, and `5 != 1`, the scores are not swapped.
- *Example 3*: At the end of a turn, the players have scores of 1 and 0; `3**(1 + 0) = 3`, and `3 == 3`, the scores are swapped.
- *Example 4*: At the end of a turn, the players have scores of 23 and 4; `3**(23 + 4) = 7625597484987`, and `7 == 7`, the scores are swapped.

Suggestions

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 framework. Also, please do **not** change any function signatures (names, argument order, or number of arguments).

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:

```
$ python hog_gui.py
```

Phase 1: Simulator

> **Important submission notes**: After completing any problems required in this phase, you need to submit your answer to `Contest 'hog: phase 1'` on our OJ website to get your answer scored. Don't forget to check the deadline on the OJ website.

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. Example: `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 writing any code, read over the `dice.py` file and check your understanding by trying the functions defined in `dice.py` in a Python interpreter and observing the results. Here is an example:

```
$ python -i dice.py
>>> i = 0
>>> while i < 10: # roll six-sided dices for ten times
...     print(six_sided())
...     i += 1
# You will see ten random values between 1 and 6 in following ten lines
_______

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

> Recall that you can use `python -i <file.py>` to start a Python interpreter that executes all statements in `<file.py>` and then enter the interactive mode where you can write call expressions to run the functions defined in `<file.py>`.

Problem 1 (200 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 (*Pig Out*).

> The Pig Out rule is reproduced below:
>
> - **Pig Out**. If any of the dice outcomes is a 1, the current player's score for the turn is 1.
>   - *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 Pig Out 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 Pig Out happens in the middle of rolling**. In this way, you correctly simulate rolling all the dice together.

> You can't implement the feral hogs rule in this problem, since `roll_dice` doesn't have the previous number of rolls as an argument. It will be implemented later.

Debug Tips

If your code cannot pass the tests in our OJ website, it's time to debug. You can observe the behavior of your function using the debugger provided by PyCharm. Also, you can using Python directly. First, start the Python interpreter and load the `hog.py` file.

```
$ python -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` 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, dice=fixed_dice)
7
```

> On most systems, you can evaluate the same expression again by pressing the up arrow on the keyboard, then pressing enter or return. If you want to get the second last, third last, etc., command you made, press up arrow repeatedly.

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

You should follow this same procedure of understanding the problem, implementing a solution, testing, and debugging for all the problems on this project.

Problem 2 (200 pt)

Implement the `free_bacon` helper function that returns the number of points scored by rolling 0 dice, based on the opponent's current score. You can assume that `score` is less than 100.

> The Free Bacon rule is reproduced below:
>
> - **Free Bacon**. A player who chooses to roll zero dice scores points equal to one more than the absolute alternating difference of the digits of the opponent's score cubed.
>   - *Example 1*: The opponent has 4 points, and the current player chooses to roll zero dice. `4*4*4 = 64`, so the current player gets `1 + |6 - 4| = 3` points.
>   - *Example 2*: The opponent has 1 points, and the current player chooses to roll zero dice. `1*1*1 = 1`, so the current player gets `1 + |1| = 2` points.

    - *Example 3*: The opponent has 20 points, and the current player chooses to roll zero dice. `20*20*20 = 8000`, so the current player gets `1 + |8 - 0 + 0 - 0| = 9` points.
    - *Example 4*: The opponent has 45 points, and the current player chooses to roll zero dice. `45*45*45 = 91125`, so the current player gets `1 + |9 - 1 + 1 - 2 + 5| = 13` points.

> The built-in `abs` function can be used.

Again, you can also test `free_bacon` interactively by entering `python -i hog.py` in the terminal and then calling `free_bacon` with various inputs.

Problem 3 (100 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 `free_bacon` when possible.

Problem 4 (200 pt)

Implement `is_swap`, which returns whether or not the scores should be swapped.

> The Swine Swap rule is reproduced below:
>
> - **Swine Swap**. Define the *excitement* of the game to be three to the power of the sum of the players' scores. After points for the turn are added to the current player's score, if the excitement's first digit and last digit are the same, the scores should be swapped.
>   - *Example 1*: At the end of a turn, the players have scores of 2 and 4; `3**(2 + 4) = 729`, and `7 != 9`, the scores are not swapped.
>   - *Example 2*: At the end of a turn, the players have scores of 11 and 1; `3**(11 + 1) = 531441`, and `5 != 1`, the scores are not swapped.
>   - *Example 3*: At the end of a turn, the players have scores of 1 and 0; `3**(1 + 0) = 3`, and `3 == 3`, the scores are swapped.
>   - *Example 4*: At the end of a turn, the players have scores of 23 and 4; `3**(23 + 4) = 7625597484987`, and `7 == 7`, the scores are swapped.

The `is_swap` function takes two arguments: the players' scores. It returns a boolean value to indicate whether the *Swine Swap* condition is met.

Problem 5a (200 pt)

Implement the `play` function, which simulates a full game of Hog. Players alternate turns rolling dice until one of the players reaches the `goal` score.

You can ignore the **Feral Hogs** rule and `feral_hogs` argument for now; You'll implement it in Problem 5b.

To determine how much 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 wants to roll in the turn. Each strategy function should be called only once per turn. Don't worry about the details of implementing strategies yet. You will develop them 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.

Here are some hints:

- You should use the functions you have already written! You will need to call `take_turn` with all three arguments.
- Only call `take_turn` once per turn.
- Enforce all the special rules except for feral hogs.
- You can get the number of the other player (either 0 or 1) by calling the provided function `other`.
- You can ignore the `say` argument to the `play` function for now. You will use it in Phase 2 of the project.

Problem 5b (100 pt)

Now, implement the Feral Hogs rule. When `play` is called **and** its `feral_hogs` argument is `True`, then this rule should be imposed. If `feral_hogs` is `False`, this rule should be ignored. (That way, test cases for 5a will still pass after you solve 5b.)

> The Feral Hogs rule is reproduced below:
>
> - **Feral Hogs**. If the number of dice you roll is exactly 2 away (absolute difference) from the number of points you scored on the previous turn, you get 3 extra points for the turn. Treat the turn before the first turn as scoring 0 points. Do **not** take into account any previous feral hog bonuses or swine swap (next rule) when calculating the number of points scored the previous turn.
>   - *Example 1*:
>   - Both players start out at 0. (0, 0)
>   - Player 0 rolls 3 dice and gets 7 points. (7, 0)
>   - Player 1 rolls 1 dice and gets 4 points. (7, 4)
>   - Player 0 rolls 5 dice and gets 10 points. 5 is 2 away from 7, so player 0 gets a bonus of 3. (20, 4)
>   - Player 1 rolls 2 dice and gets 8 points. 2 is 2 away from 4, so player 1 gets a bonus of 3. (20, 15)
>   - Player 0 rolls 8 dice and gets 20 points. 8 is 2 away from 10, so player 0 gets a bonus of 3. (43, 15)
>   - Player 1 rolls 6 dice and gets 1 point. 6 is 2 away from 8, so player 1 gets a bonus of 3. (43, 19)
>   - *Example 2*:
>   - Both players start out at 0. (0, 0)
>   - Player 0 rolls 2 dice and gets 3 points. 2 is 2 away from 0, so player 0 gets a

> bonus of 3. (6, 0)

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:

```
$ python 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.

Congratulations! You have finished Phase 1 of this project!

Phase 2: Commentary

> **Important submission notes**: After completing any problems required in the following two phases, you need to submit your answer to `Contest 'hog: phase 2&3'` on our OJ website to get your answer scored. Don't forget to check the deadline on the OJ website.

In the second phase, you will implement commentary functions that print remarks about the game after each turn, such as, `"22 points! That's the biggest gain yet for Player 1."`

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(prev_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 != prev_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.

    NOTE: the following game is not possible under the rules, it's just
    an example for the sake of the doctest

    >>> 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, 6)
    Player 0 now has 10 and Player 1 now has 6
    >>> h3 = h2(6, 17)
    Player 0 now has 6 and Player 1 now has 17
    Player 1 takes the lead by 11
    """
    def say(score0, score1):
        return both(f(score0, score1), g(score0, score1))
    return say
```

Problem 6 (200 pt)

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.

Problem 7 (300 pt)

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. E.g., `announce_highest(1)` and its return value ignore Player 0 entirely and just print information about Player 1. To compute the gain, it must compare the score from last turn to the score from this turn for the player of interest, which is designated by the `who` argument. This function must also keep track of the highest gain for the player so far.

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.

> Note: The doctests for both/announce_highest in hog.py might describe a game that can't occur according to the rules. This shouldn't be an issue for commentary functions since they don't implement any of the rules of the game.

> 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`.

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

```
$ python 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!

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 (200 pt)

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

```
>>> def printed(g):
...     def print_and_return(*args):
...         result = g(*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.

Problem 9 (200 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`.

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.

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

```
$ python 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(6)`.

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

Problem 10 (100 pt)

A strategy can try to 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.

Note it is impossible for strategies to know what number of points the current player earned on the previous turn, and thus we cannot predict feral hogs. For strategies, we do not take into account bonuses from feral hogs to calculate bonuses against the margin or whether a swap will occur

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 (200 pt)

A strategy can also take advantage of the *Swine Swap* rule. The swap strategy always rolls 0 if doing so triggers a beneficial swap and always avoids rolling 0 if doing so triggers a detrimental swap. In other cases, it rolls 0 if rolling 0 would give **at least** `margin` points. Otherwise, the strategy rolls `num_rolls`.

> Note it is impossible for strategies to know what number of points the current player earned on the previous turn, and thus we cannot predict feral hogs. For strategies, we do not take into account bonuses from feral hogs to calculate bonuses against the margin or whether a swap will occur
>
> Hint: a tie is technically a "swap" (e.g., 43 being swapped with 43), but is considered neither detrimental nor beneficial for the purposes of this problem.

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(4)`.

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(4)` strategy. Some suggestions:

- `swap_strategy` is a good default strategy to start with.
- There's no point in scoring more than 100. 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 a beneficial swap rolling more than 0 dice.
- Choose the `num_rolls` and `margin` arguments carefully.
- Take the action that is most likely to win the game.

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

```
$ python hog_gui.py
```

The GUI will alternate which player is controlled by you.

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