Introduction to Artificial Intelligence
Problem Set 1 — Coding

Instructions

Welcome to Intro to Artificial Intelligence! In this first problem set, we'll be going over some foundational programming skills that will be very important for the homework assignments later on in the semester. For some students this will be a quick review, for others this material might be completely new, both cases are fine. If this is your first time working with Python, or if you've seen or even used Python before but it's been a while, some of these concepts may seem a bit peculiar, and that's OK. We're spending time on some very specific techniques and unique Python oddities now to avoid being confused by them later on during the homework assignments. Don't worry if you get stuck on some questions, but do

- Bring your questions to class and to office hours. Don't wait until you get stuck on a complicated part of a homework somewhere down the line.
- Attempt to complete as much as you can, and
- **TURN IN YOUR WORK before the due date**. You won't be penalized for incomplete work, or for errors in your code or logic, but you won't get any credit if you don't turn anything in. This is true for all problem set assignments in this course.

If all of these questions seem easy and obvious, try to challenge yourself:

- See if you can write the code to answer each question in the fewest number of lines possible
- Run some experiments to see how efficient these recursive implementations are (particularly with the last few questions)
- Try to generalize the `memoizer` function.

Regardless of your experience with Python you should be able to get something out of this exercise.

Questions

1. The Collatz conjecture[1] is a deep mathematical question that can be explained with a fairly simply game of arithmetic:
   (a) Pick a positive integer, call it n.
   (b) If n is even, divide it by 2.
   (c) Otherwise, multiply it by 3 and add 1.
   (d) Repeat steps 2-4 with the new number.

[1] `https://en.wikipedia.org/wiki/Collatz_conjecture`

The mathematical question is: "Does every starting n eventually end up at 1?" if you keep applying the rules over and over again (notice that if you start at 1, you go to 4 then 2 and then end up right back at 1, you're caught in a loop). Write a function in Python which, when given any integer n, runs the game above until 1 is reached, printing out all the numbers visited along the way. While it may seem like the obvious answer is an iterative solution, we're going to go with a recursive version. We'll see why shortly.

2. After you've got the basic game working, lets start really taking advantage of the fact that a computer can do this process far quicker than we can by hand. While the numbers we visit along the way can be interesting and aesthetically pleasing, there are other questions we could answer. Write another function which **returns** (not prints) the number of steps it takes to go from n to 1. This should also be recursive.

3. You may have wondered why we're using recursion for these functions, rather than straightforward loops. One reason (we'll give another reason soon) is that practicing recursion is partly the purpose of the exercise: understanding recursion and being comfortable implementing things recursively is going to be very important for future homework assignments. But it is true that recursive code has some drawbacks: notably, you can fill up the call-stack with recursive calls, which can cause your code to crash. You've probably encountered this when debugging a faulty base-case, but it can happen even when there are no logical errors in your code. One way to address this is to use *memoization*[2]. Write a function called `memoizer` which takes as its arguments a function, and a storage dictionary, and returns a **new function**, a "memoized" version of the function that was passed in. You can assume that the function-to-be-memoized takes only positional arguments (no keyword args). If you've never written Python functions that have variable or keyword arguments, or take functions as arguments, you're going to have to do a little bit of research on these features[3][4]. Once you've got this written, use the `memoizer` to make a memoized version of your previous two functions. They should function exactly the same, but we'll be able to compute things much more quickly in the next few questions.

4. Next up are two functions which will find for us values for n which take the longest to get back to 1. Write `max_collatz_steps` and `max_collatz_num` which return the largest number of steps, and the corresponding n out of all the numbers from 1 up to the passed in value. Using our memoized collatz functions, we can now compute the maximum values for a fairly large range of numbers. Think about whether this would have been more or less efficient with a non-recursive solution, and test out your hypothesis by running experiments with iterative vs recursive (memoized) functions. If you're feeling ambitious, think about how you might take advantage of Python's rebinding capabilities to eke out even more efficiency.

5. Visualizing the chains or "orbits" of numbers in this process can result in some remarkable images (see wikipedia link in the first footnote), but to make these visualizations we'd need to keep track of the *path* that we took along the way. Modify (or write a new function) which builds up a graph linking numbers to their "Collatz successor". The Object Oriented way to do this would be to write our own `Node` class, but really we don't need to do that: we can just use Python's built in dictionaries as our nodes. This is a somewhat more "Pythonic" approach for when we need an object that only holds on to data (rather than one that has useful methods), and we'll be using it in future homework assignments, so it's a good idea to practice it here.

Submission

You can keep all of your code in a single python file, and upload it to Gradescope as your submission for this assignment. If you ran additional experiments or visualizations (completely optional), put these and

[2] `https://en.wikipedia.org/wiki/Memoization`
[3] `https://docs.python.org/3/reference/`,
[4] `https://docs.python.org/3/library/`

any discussion in a single PDF document and include it as well. Otherwise, include any notes or discussion as comments in your python file.