Lab: Recursion and Memoization

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

Day 1

The Fibonacci function has the following form:

\[
f(n) = \begin{cases} 
1 & n = 0 \text{ or } n = 1 \\ 
f(n-1) + f(n-2) & \text{else.} 
\end{cases}
\]

This can be written in python as:

```python
def f(n):
    if n == 0 or n == 1:
        return 1
    return f(n-1) + f(n-2)
```

You work as a researcher combatting the zombie virus that is infecting American cities. At day \( t = 0 \), patient zero moves to Missoula from a large metropolis. In each day, a zombie bites two more people, making a new zombie, and so the rate of infection grows with the number of infected. Furthermore, you find that zombies that have been alive for a day or longer tend to meet up with one another and produce zombie offspring; however, zombies die on their third day. Where \( z(t) \) is the number of zombies on day \( t \), your model predicts

\[
z(t) = \begin{cases} 
1 & t = 0 \text{ or } t = 1 \text{ or } t = 2 \\ 
2 \cdot z(t-1) + 3 \cdot z(t-2) - z(t-3) & \text{else.} 
\end{cases}
\]

1. Write a recursive function \( z(t) \) that computes the number of zombies living at day \( t \).

2. Compute and print \( z(20) \), the number of zombies alive at day 20. Verify that this is 279831760.

3. Use a for loop to compute the first day where Missoula becomes all zombies: i.e., loop through \( t \) to find the first \( t \) for which \( z(t) > 73340 \).
Day 2

The time package lets us compute the elapsed time of some python code. For example, to run a function $f(10)$ and another function $g(20)$ and print the results, we can compute the runtime as follows:

```python
import time
t1=time.time()
print f(10)
print g(20)
t2=time.time()
print 'This took', t2-t1, 'seconds'
```

1. Use the time package to print the number of elapsed seconds to compute $z(30)$. How long did it take to run?

2. “Memoizing” let’s us avoid computations we’ve already done by using a dictionary. This is done by including the following code and then putting `@Memoized` above any function to memoize it (this is called a “decorator”):

```python
class Memoized:
    def __init__(self, function):
        self._function = function
        self._cache = {}
    def __call__(self, *args):
        if args not in self._cache:
            # not in the cache: call the function and store the result in
            # the cache
            self._cache[args] = self._function(*args)
        # the result must now be in the cache:
        return self._cache[args]
```

You can use this decorator as follows:

```python
@Memoized
def some_function(args):
    # write your function and it will automatically be memoized!
    pass
```
3. Create a memoized version of the Fibonacci function $f$ using the \texttt{@Memoized} decorator. How long does it take to compute $f(40)$?

4. Create a memoized version of the zombie function $z$ using the \texttt{@Memoized} decorator. How long does it take to compute $z(30)$? What is the speedup of that over the non-memoized runtime you computed?

5. \textbf{[BONUS]} Create a manually memoized version of the Fibonacci function by making a cache called \texttt{n_to_fibonacci}, in which you store all computed solutions.

6. \textbf{[BONUS]} Create a manually memoized version of the zombie function using a cache called \texttt{t_to_zombies}, in which you store all computed solutions.

Note that if your memoized function goes too deep in the recursions, python may abort execution out of fear that too much memory will be used. You can override this by writing the following at the top of your program:

```python
import sys
sys.setrecursionlimit(2000)
```

You can replace 2000 with a larger value until your program does not crash (but beware that if you forgot the base case in your recursive function, your computer may run out of RAM).