CS 331 – Assignment 7

Due: 11:59pm Monday, May 5

Your starting point for this assignment is the code snippet file from April 23 in which we defined the JavaScript Sequence object constructor with a non-memoized take function.

When you've completed your work on the assignment, upload the new Sequence object, along with your code for problems 4, 5, and 6 in a file called *assign7.js* to the ASSIGNMENT 7 dropbox on D2L. We will test your code as follows:

```
$ node
> .load assign7.js
⋮
                                   Various test cases including all of those in the samples below
⋮
```

1. Add a map function to the Sequence object that returns a new Sequence resulting from the application of a function f to each member of the original Sequence. Example of its use:

```
> fib = function() {
    var x = 1;
    var y = 1;
    return function() {
        var prev = x;
        x = y;
        y += prev;
        return prev;
    };
};
b = new Sequence( fib() );
c = b.map(function (x) {return x+1;});
⋮
                                                    "undefined" returned here from node
>  c.take(10)
[ 2, 2, 3, 4, 6, 9, 14, 22, 35, 56 ]
```

2. The Sequence object as you have used it in the preceding problem is both flawed and inefficient. To understand why it is flawed, consider the following:

```
> fib = function() {
    var x = 1;
    var y = 1;
    return function() {
        var prev = x;
        x = y;
        y += prev;
        return prev;
    };
};
b = new Sequence( fib() );
> b.take(10)
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
> b.take(10)
[ 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765 ]
```

   Fortunately you can fix the flaw and at the same time make the Sequence object vastly more efficient by *memoizing* the lazy evaluation in the Sequence. You will then have what is known as call-by-need evaluation instead of call-by-name evaluation.

   Example of how the take function should work after you memoize the evaluation:

```
> b.take(10)
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
> b.take(20)
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765 ]
```

3. Add a filter function to the Sequence object that returns a new Sequence comprising all members of the original sequence that pass a boolean-valued test that is given to the filter. Example of its use:

```
ints_from1 = function() {
    var x = 1;
    return function() {
        var temp = x;
        x = x + 1;
        return temp;
    };
};
y = new Sequence( ints_from1() ).filter(function (x) { return x % 2 === 0; });
```

⋮

"undefined" returned here from node

```
> y.take(10)
[ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ]
```

4. For this problem, write a function that generates thunks called *iterates_from*. *iterates_from* takes an initial term s for a sequence and function f. When called, it produces the thunk to generate the sequence:

$$s, f(s), f(f(s)), f(f(f(s))), \ldots$$

Example of usage:

```
> powers_of_two = iterates_from(1, function (x) { return 2 * x; } );
q = new Sequence(powers_of_two());
```

⋮

"undefined" returned here from node

```
> q.take(10)
[ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 ]
>
```

5. In this problem, use one or more of the functions you've developed in Problems 1, 3, or 4 to generate "hailstone sequences". The *hailstone sequence starting at n* is defined as follows. Starting with n, form the sequence by the following rule:

- If n is even, divide it by 2 to give $n' = n/2$.
- If n is odd, multiply it by 3 and add 1 to give $n' = 3n + 1$.

Then take n' as the new starting point and repeat the process. An unproven conjecture is that, no matter what starting point is used, all hailstone sequence eventually settle into the repeating pattern 4, 2, 1, 4, 2, 1 ...

Use your solution to this problem to bind the hailstone sequence starting at 5 to the JavaScript variable *hailstone5* and the hailstone sequence starting at 11 to the JavaScript variable *hailstone11*. Thus, when we test your code, we should see:

```
> hailstone5.take(10)
[ 5, 16, 8, 4, 2, 1, 4, 2, 1, 4 ]
> hailstone11.take(10)
[ 11, 34, 17, 52, 26, 13, 40, 20, 10, 5 ]
```

6. Write a function called *interleave* that takes two sequences and returns a new sequence whose members are the members of the original sequences interleaved with each other. That is, given two sequences:

$$x_0, x_1, x_2, x_3, \ldots \text{ and } y_0, y_1, y_2, y_3, \ldots$$

interleave returns the sequence:

$$x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3 \ldots$$

Example of use:

```
> interleave(hailstone11,hailstone5).take(20)
[ 11, 5, 34, 16, 17, 8, 52, 4, 26, 2, 13, 1, 40, 4, 20, 2, 10, 1, 5, 4 ]
```