Assignment #1—Simple JavaScript Programs

Due: October 4th, 2019 at 5:00pm

Your job in this assignment is to write programs to solve five programming problems. The starter code is a zip file which when expanded produces five folders, one per problem. Each folder contains an HTML file that can more or less be ignored, except that you need to double-click it to load it into a browser (we recommend Chrome). You’re to modify each JavaScript file in a simple, JavaScript-aware editor (we recommend Atom or Sublime). As you make changes to your JavaScript files, save and reload the companion HTML page to see how what you’ve written is working.

Problem 1: Validating Credit Card Numbers

When creditors like Visa, MasterCard, and American Express issue new credit cards, they ensure the numbers are valid according to Luhn’s algorithm. Luhn’s algorithm can then be used by online retailers to immediately reject most mistyped or intentionally manufactured numbers.

Luhn’s algorithm is a digit manipulation algorithm that works like this:

- Isolate every digit, starting from the right and moving left, doubling every second one. When this doubling produces a value greater than 9, subtract 9 from it. For example, 596825 would produce:
  
  - 5 on behalf of the 5 in the ones place
  - 2 * 2 = 4 on behalf of the 2 in the tens place
  - 8 on behalf of the 8 in the hundreds place
  - 2 * 6 - 9 = 3 on behalf of the 6 in the thousands place
  - 9 on behalf of the 9 in the ten thousands place
  - 2 * 5 - 9 = 1 on behalf of the 5 in the hundred thousands place

- Sum all of the transformed digits to produce the Luhn digit sum. For example, the 596825 above has a Luhn digit sum of $5 + 4 + 8 + 3 + 9 + 1 = 30$.

- If the Luhn digit sum ends in a 0, then and only then is the original number valid according to Luhn’s algorithm. For example, 596825 is technically a valid credit card number according to Luhn’s algorithm, whereas 596725 (where there’s a 7 in place of the 8) is not.

Update the Luhn.js file to include your own implementation of the predicate function called isValid, which returns true if the credit card number looks to be valid according to Luhn’s algorithm, and false otherwise. Launch luhn.html to exercise your implementation using the supplied unit tests and any additional ones you want to
Problem 2: Hailstone Sequences

Douglas Hofstadter’s Pulitzer-prize-winning book Gödel, Escher, Bach contains many interesting mathematical puzzles, many of which can be expressed in the form of computer programs. Of these, most require programming skills well beyond the second week of CS 106AX. However, in Chapter XII, Hofstadter mentions a wonderful problem that is well within the scope of the control statements we reviewed this past week. The problem can be expressed as follows:

Pick some positive integer and call it \( n \).
If \( n \) is even, divide it by two.
If \( n \) is odd, multiply it by three and add one.
Continue this process until \( n \) is equal to one.

On page 401 of the Vintage edition, Hofstadter illustrates this process with the following example, starting with the number 15:

\[
\begin{align*}
15 & \text{ is odd, so I make } 3n+1: & 46 \\
46 & \text{ is even, so I take half:} & 23 \\
23 & \text{ is odd, so I make } 3n+1: & 70 \\
70 & \text{ is even, so I take half:} & 35 \\
35 & \text{ is odd, so I make } 3n+1: & 106 \\
106 & \text{ is even, so I take half:} & 53 \\
53 & \text{ is odd, so I make } 3n+1: & 160 \\
160 & \text{ is even, so I take half:} & 80 \\
80 & \text{ is even, so I take half:} & 40 \\
40 & \text{ is even, so I take half:} & 20 \\
20 & \text{ is even, so I take half:} & 10 \\
10 & \text{ is even, so I take half:} & 5 \\
5 & \text{ is odd, so I make } 3n+1: & 16 \\
16 & \text{ is even, so I take half:} & 8 \\
8 & \text{ is even, so I take half:} & 4 \\
4 & \text{ is even, so I take half:} & 2 \\
2 & \text{ is even, so I take half:} & 1
\end{align*}
\]

As you can see from this example, the numbers go up and down, but eventually—at least for all numbers that have ever been tried—comes down to end in 1. In some respects, this process is reminiscent of the formation of hailstones, which get carried upward by the winds over and over again before they finally descend to the ground. Because of this analogy, this sequence of numbers is usually called the Hailstone sequence, although it goes by many other names as well.

Write a function `hailstone` that takes an integer and then uses `console.log` to display the Hailstone sequence for that number, just as in Hofstadter’s book, followed by a line showing the number of steps taken to reach 1. For example, your program should be able to produce a output that looks like this when `hailstone(17)` is called:
Aside: One fascinating thing about this problem is that no one has yet been able to prove that it always stops. The number of steps in the process can certainly get very large. How many steps, for example, does your program take when \( n \) is 27? The conjecture that this process always terminates is called the **Collatz conjecture**, and appears in the **XKCD** cartoon by Randall Munroe on the right.

\[
\begin{align*}
17 & \text{ is odd, so I make } 3n+1: 52 \\
52 & \text{ is even, so I take half: 26} \\
26 & \text{ is even, so I take half: 13} \\
13 & \text{ is odd, so I make } 3n+1: 40 \\
40 & \text{ is even, so I take half: 20} \\
20 & \text{ is even, so I take half: 10} \\
10 & \text{ is even, so I take half: 5} \\
5 & \text{ is odd, so I make } 3n+1: 16 \\
16 & \text{ is even, so I take half: 8} \\
8 & \text{ is even, so I take half: 4} \\
4 & \text{ is even, so I take half: 2} \\
2 & \text{ is even, so I take half: 1} \\
\text{The process took 12 steps to reach 1.}
\end{align*}
\]
Problem 3: Stern-Brocot Trees and Sequences

This exercise involves the following construction, which is an adaptation of something known as the Stern-Brocot tree:

Each fraction is \( \frac{n_L + n_R}{d_L + d_R} \), where \( \frac{n_L}{d_L} \) is the closest ancestor up and to the left, and \( \frac{n_R}{d_R} \) is the closest ancestor up and to the right. For example, \( \frac{3}{7} \), for example, is produced from \( \frac{2}{5} \) (first ancestor up and to the left) and \( \frac{1}{2} \) (first ancestor up and to the right.)

This manner of enumerating fractions has three interesting properties (stated without proof, but you can trust that they’re correct):

- each fraction generated by the construction is in reduced form,
- every single reduced fraction between 0 and 1 will eventually be formed, and
- \( \frac{n_L}{d_L} \) is always less than \( \frac{n_L + n_R}{d_L + d_R} \), and \( \frac{n_L + n_R}{d_L + d_R} \) is always less than \( \frac{n_R}{d_R} \).

Each rational number between 0 and 1 can be expressed as a sequence of R’s and L’s. These R’s and L’s signal how one should descend the Stern-Brocot tree from \( \frac{1}{2} \) to arrive at the fraction it represents. \( \frac{1}{2} \) is represented by the empty sequence, \( \frac{1}{3} \) is represented by L, and \( \frac{3}{7} \) is represented by LRR. Whenever a letter appears two or more times in a run, we compress that run so that something like LLRLRRRLLL is instead represented as L2 R L R5 LL. Irrational numbers like \( \pi \) - 3 don’t appear in the tree, but rational numbers close to them do! If we keep descending through the tree until we hit some
maximum sequence length, then we use that Stern-Brocot sequence and just say that it’s good enough.

For this problem, you’re to implement a function called sbs (that’s short for Stern-Brocot sequence) that accepts a positive, real number less than 1 and returns the Stern-Brocot sequence of R’s and L’s as outlined above. (You’ll receive full credit if you generate uncompressed strings like LLRRRLLLLLRRRLR instead of L2 R3 L5 R2 L R, though we’ll give you extra credit if you generate the compressed form.) An optional second argument specifies the maximum number of Rs and Ls in the sequence; if that second argument is missing, it defaults to 500. For instance, the following test harness would produce the specified output, provided the implementation of sbs is solid:

```javascript
function TestSternBrocotSequences() {
    console.log("sbs(0.5) -> "+sbs(0.5));
    console.log("sbs(0.125) -> "+sbs(0.125));
    console.log("sbs(0.65) -> "+sbs(0.65));
    console.log("sbs(Math.E - 2) -> " + sbs(Math.E - 2));
    console.log("sbs(Math.PI - 3) -> " + sbs(Math.PI - 3));
    console.log("sbs(Math.PI - 3, 100) -> " + sbs(Math.PI - 3, 100));
}
```

This is the algorithmically most intense function you need to write for Assignment 1, so don’t be shy asking for help if you get stuck.
Problem 4 (Chapter 4, exercise 4, page 148)
Use the **GObject** hierarchy of classes to draw a rainbow that looks much like this:

![Rainbow Image](image-url)

Starting at the top, the seven bands in the rainbow are red, orange, yellow, green, blue, indigo, and violet, respectively; cyan makes a lovely color for the sky. Note that Chapter 3 defines the **GRect** and **GOval** classes and does not include a graphical object to represent an arc. It will help to think outside the box, in a more literal sense than usual.

Rather than specify the exact dimensions of each circle (and there are indeed circles here), play around with their sizes and positioning until you get something that matches your aesthetic sensibilities. The only things we’ll truly require are:

- The top of the arc should not be off the screen.
- Each of the arcs in the rainbow should get clipped along the sides of the window, and not along the bottom.

Place your implementation in a file called **Rainbow.js**, and test your implementation by loading and reloading **rainbow.html** in your browser.

Problem 5: Sampler Quilt (based on Chapter 4, exercise 16, page 154)
Back in the early 1990s—long before JavaScript existed—Julie Zelenski and Katie Capps Parlante developed a graphics assignment that we used in our introductory courses for several years, and I’ve decided to revive it. 😊 The goal of the assignment was to draw a sampler quilt, which is composed of several different block types that illustrate a variety of quilting styles.

The sampler quilt should ultimately look like that presented at the top of the next page.
The quilt is $7 \times 7$, there are four different patch types, and the $k^{th}$ row starts with the $k^{th}$ patch type and rotates cycles through the others, repeating as necessary.

- The bullseye patch is seven evenly spaced concentric circles alternating between red and white. The borders of the circle, like all borders of all shapes in the quilt, is black.
- The log cabin patch draws several four, square frames of dovetailed logs—or rather, tan rectangles—such that the shorter dimension of each log equals the dimension of the open center square.
• The flower patch consists of five circles, all filled with randomly generated colors using the `randomColor()` function discussed in the textbook. The radius of each circle is one fifth the dimension of the patch itself, the fifth circle is centered in the patch and layered on top of the other four, each of which is centered in the patch’s four main quadrants.

• The final patch—the i-love-my-section-leaders patch—consists of a randomly selected image (with a URL of

  https://cs106ax.stanford.edu/img/ryan-75x75.jpg.

where `ryan` is equally likely to be `ryan`, `esteban`, `suzanne`, `anand`, or `jonathan`). The images are all exactly the size needed to fit snugly within a patch, though you layer two frames—the outer one pink, and the inner one white—on top of the image to make it look like a photo that’s sitting at home in your family’s living room. You’ll want to investigate the `GImage` class discussion in the textbook, and you’ll want to leverage log cabin patch code you wrote to get the pink and white photo framing.

The point of the exercise is to come up with a clean decomposition and implementation that sensibly reuses code wherever possible. We don’t care about the exact pixel alignments so much as we do code clarity and narrative. Don’t sweat details that don’t feel pedagogically important. And if you have questions, just ask us and we’ll answer them.