Jerry Cain
CS 106AJ

Handout #31S
October 31, 2018

Solutions for Section #5

Solution 1: String Split

Some thought questions to ensure you understand the solution:

- Why does the **`for`** loop test rely on **`<=`** instead of <?
- What's the best description you have for what `i` is tracking on behalf of the algorithm?
- The internal **`if`** test checks to see if **`i === str.length`** first before advancing on to check the return value of **`indexOf`**?

```
/**
 * Function: split
 * ---------------
 * Returns an array of the spplied string when exploded around
 * all of the characters within the supplied delimiter.
 */
function split(str, delimiters) {
   let start = 0;
   let fragments = [];
   for (let i = 0; i <= str.length; i++) {
      if (i === str.length || delimiters.indexOf(str.charAt(i)) !== -1) {
         let fragment = str.substring(start, i);
         fragments.push(fragment);
         start = i + 1;
      }
   }
   return fragments;
}
```

Solution 2: Keith Numbers

Some thought questions to ensure you understand the solution:

- What does the use of array throughout the implementation of **`isKeithNumber`** buy you? What would have been the alternative?
- How would the implementation of **`isKeithNumber`** need to change had the implementation of **`createDigitsArray`** not reversed the digits array just before returning it?
- What's the advantage of calling **`shift`** on the **`partials`** array within **`isKeithNumber`**? Had the shift call been omitted, how could the implementation of **`isKeithNumber`** change to account for the omission?
- Note that the while loop test within **`isKeithNumber`** uses **`<`** instead of **`<=`**. What would have happened had you accidentally used **`<=`** instead?

```
/**
 * Predicate Function: isKeithNumber
 * ---------------------------------
 * Returns true if and only if the supplied integer,
 * assumed to be positive, is a Keith number.
 *
 * It does so by maintaining as much of the Fibonacci-like
 * sequence needed to generate the next sequence number,
 * and stops when the most recently introduced number either
 * equals n (that's good!) or exceeds it (that's not good!)
 */
function isKeithNumber(n) {
   if (n <= 0) return false;
   let partials = createDigitsArray(n);
   while (partials[partials.length - 1] < n) {
      let sum = sumArray(partials);
      partials.push(sum);
      partials.shift();
   }
   return partials[partials.length - 1] === n;
}

/**
 * Function: createDigitsArray
 * ---------------------------
 * Accepts an integer called n (assumed to be positive)
 * and produces an array of all of its digits, in order,
 * such that the most significant digit is in the leading
 * position and the least significant digit is in
 * the final position.
 */
function createDigitsArray(n) {
   let digits = [];
   while (n > 0) {
      let digit = n % 10;
      digits.push(digit);
      n = Math.floor(n/10);
   }
   digits.reverse();
   return digits;
}

/**
 * Function: sumArray
 * ------------------
 * Returns the sum of all integers residing with the
 * supplied array.
 */
function sumArray(array) {
   let sum = 0;
   for (let i = 0; i < array.length; i++) {
      sum += array[i];
   }
   return sum;
}
```

Solution 3: Disappearing Squiggles

```
/**
 * File: DisappearingSquiggles.js
 * ------------------
 * This graphics program allows a user to draw squiggles that,
 * once completed, live for five seconds before disappearing.
 */

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const DELAY = 5000;

/**
 * Function: DisappearingSquiggles
 * -------------------------------
 * Implements the full graphics program that allows users to
 * draw squiggles that disappear after five seconds.
 */
function DisappearingSquiggles() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   let inProgress = null;      // no squiggle actively being drawn
   let lastx = -1, lasty = -1; // no squiggle actively being drawn

   /**
    * Inner function: mousedownAction
    * -------------------------------
    * Initiates the squiggling process by noting that
    * no lines have been drawn just yet while recording
    * the position of the mousedown event so the first
    * drag event knows where the user first clicked.
    */
   let mousedownAction = function(e) {
      inProgress = [];
      lastx = e.getX();
      lasty = e.getY();
   };

   /**
    * Inner function: dragAction
    * --------------------------
    * Lays down a line between the most recent mouse
    * event location (either the first location from
    * mousedownAction, or from the previous dragAction),
    * caches the line that was just drawn in an array that
    * can easily be reached during erase time, and records
    * the current mouse drag location so the *next* drag
    * action knows where the next line to be drawn starts.
    */
   let dragAction = function(e) {
      let line = GLine(lastx, lasty, e.getX(), e.getY());
      gw.add(line);
      inProgress.push(line);
      lastx = e.getX();
      lasty = e.getY();
   };
```

```
   /**
    * Inner function: mouseupAction
    * -----------------------------
    * Takes a snapshot of all the lines that have accumulated
    * since the last mousedown event, since those all contribute
    * to the very squiggle that needs to be erased five seconds
    * from now.
    */
   let mouseupAction = function(e) {
      let completed = inProgress; // thought question: why is this necessary?
      let removeSquiggle = function() {
         for (let i = 0; i < completed.length; i++) {
            gw.remove(completed[i]);
         }
      };
      setTimeout(removeSquiggle, DELAY);
      // next three lines are technically not necessary,
      // but good for bookkeeping purposes
      inProgress = null;
      lastx = -1;
      lasty = -1;
   }

   gw.addEventListener("mousedown", mousedownAction);
   gw.addEventListener("drag", dragAction);
   gw.addEventListener("mouseup", mouseupAction);
}
```