Jerry Cain
CS 106AJ

Handout #31
October 29, 2018

Section Handout #5

Problem 1: String Split

JavaScript's String class includes a **`split`** method that accepts a separator string (e.g. **`" "`** or **`"sh"`**) and splits the receiving string into an array of substrings using the provided separator to determine where to make each split. Here are some examples illustrating how this built-in **`split`** method works:

```
"this is how split works".split("i") ⇒ ["th","s ","s how spl","t works"]
        "abracadabra".split("a") ⇒ ["", "br", "c", "d", "br", ""]
              "sheepishly".split("sh") ⇒ ["", "eepi", "ly"]
```

Note that the last example above splits on a separator that's more than a single character. And when the separator appears at the beginning and end of the receiving string, the first and last entries of the split are empty strings.

There are scenarios, however, where you want to split around all of many different single-character separators, not just one. A top-level function called **`split`** —which doesn't exist in JavaScript, so you'll need to implement it yourself—might operate like this:

```
split("abcdefghijklmnopqrstuvwxyz", "aeiou") ⇒
    ["","bcd","fgh","jklmn","pqrst","vwxyz"]
split("abracadabra", "abcd") ⇒ ["","","r","","","","","","r",""]
```

The separator string is really a set of many single-character strings, and each single-character string is a separator. The first of the two examples above splits around the five lowercase vowels, and the seconds splits a word around the first four letters of the lowercase alphabet.

Leverage your understanding of strings and Wednesday's introduction to arrays, implement the top-level function called **`split`**.

```
function split(str, separators) {
```

Problem 2: Keith Numbers

A Keith number is a number that appears in a Fibonacci-like series with initial terms based on its own digits. For example, 197 is a Keith number, and here's proof:

1, 9, 7, 17, 33, 57, 107, 197

When asking whether or not 197 is a Keith number, you launch a sequence using its digits, and extend the sequence so that each one beyond the first three is equal to the sum of the three that precede it.

1, 9, and 7 are the initial numbers.
17 comes next, because 1 + 9 + 7 equals 17.
33 comes next, because 9 + 7 + 17 equals 33.
57 comes next, because 7 + 17 + 33 equals 57.
107 comes next, because 17 + 33 + 57 equals 107.
197 comes next, because 33 + 57 + 107 equals 197.

Whenever the original number eventually appears in the relevant sequence, we call the number a Keith number.

Because 34285 appears in the sequence spawned from its digits, it too is a Keith number. But because it's a five-digit number, each term in the sequence is the sum of the preceding five (not three) numbers.

3, 4, 2, 8, 5, 22, 41, 78, 154, 300, 595, 1168, 2295, 4512, 8870, 17440, 34285

For this problem, write two JavaScript functions:

1. First, implement a function called **`createDigitsArray`**, which accepts a positive integer n and returns an array populated with the digits of n, in order. So, if given the number 73910, an array of length 5 would be returned, and the numbers 7, 3, 9, 1, and 0 would occupy positions 0, 1, 2, 3, and 4, respectively.

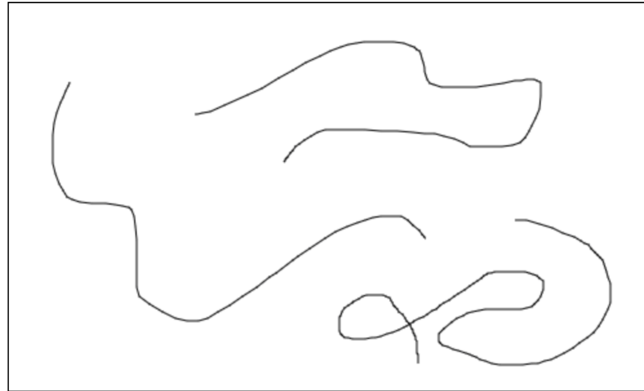
```
function createDigitsArray(n) {
```

2. Now, using your **`createDigitsArray`** function, implement the **`isKeithNumber`** predicate function, which takes a positive integer and builds the Fibonacci-like sequence up to include one number greater than or equal to the one supplied as an argument. Once that's done, you have just the right amount of information you need to return **`true`** if the original number is a Keith number, and **`false`** if it isn't.

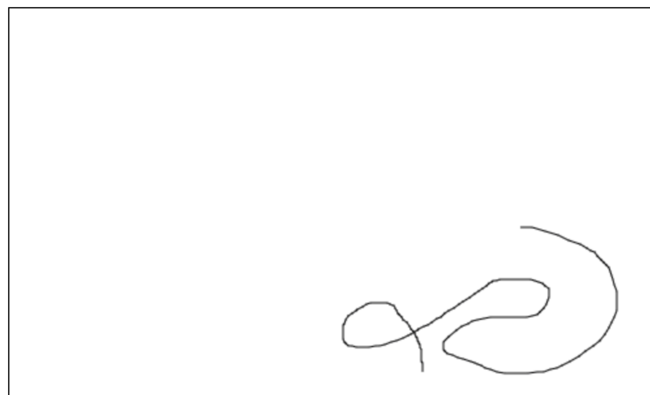
```
function isKeithNumber(n) {
```

Problem 3: Disappearing Squiggles

Write a full program that relies on **`mousedown`**, **`drag`**, and **`mouseup`** mouse events to allow the user to draw disappearing squiggles within the graphics window, as this below:



The **`mousedown`** event initiates the process of drawing a squiggle, the accumulation of drag actions that follow lay down a corresponding accumulation of many small lines that connect all of the drag event locations, and the **`mouseup`** action stops drawing and schedules a squiggle-erasing function to be executed five seconds later. If, for example, the three squiggles above are completed at t = 6, 9, and 13 seconds, at t = 16 seconds, graphics window would look like this (assuming the remaining squiggle was the last to be drawn):



This problem is architecturally similar to the line drawing program we presented in lecture when we first taught you about mouse events. Arrays come in, however, because you need to remember all of the small lines that contributed to any one squiggle so that those lines can be collectively erased five seconds post **`mouseup`**, even if the user quickly advances to draw additional squiggles.