Jerry Cain
CS 106AX

Handout #09S
October 20, 2023

Solutions for Section #3

Section solution by Jerry Cain.

Solution 1: String Split

```
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;
}
```

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?
- Why does the **`if`** test check to see if **`i === str.length`** first before advancing on to check the return value of **`indexOf`**?

Solution 2: Strings, Arrays, and Disguised Algorithms

a) The provided code—batty variable names notwithstanding—is an implementation of Kadane's algorithm, which uses a technique called dynamic programming to compute the largest subarray sum in an array of integers.

`perplexity([-2, 1, -3, 4, -1, 2, 1, -5, 7, -10]);`

produces the following

-2	1	1	4	4	5	6	6	8	8

b) The provided code is a key contribution to the implementation of the Knuth-Morris-Pratt algorithm, which works to find a particular substring within a larger string. In particular, **`result[i]`** stores the length of the longest proper prefix of **`str.substring(0, i + 1)`** that's also a proper suffix. **`result[9]`**, for instance, say that the first four characters of **`str`** match the last four characters of **`str`**.

0	1	2	0	1	2	3	3	3	4

Solution 3: Keith Numbers

```
/*
 * 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 (yay!) or exceeds it (opposite of yay!)
 */
function isKeithNumber(n) {
   if (n <= 0) return false;
   let partials = createDigitsArray(n);
   while (partials[partials.length - 1] < n) {
      let sum = sumArray(partials); // see Lecture 08 slides
      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;
}
```

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?

Solution 4: RNA, Codons, and Data Structures

```
/**
 * Predicate Function: mappingIsValid
 * ----------------------------------
 * Returns true if the supplied gene is a valid encoding
 * of the supplied amino acid sequence, and false otherwise.
 */
function mappingIsValid(gene, sequence) {
   if (gene.length !== 3 * (sequence.length + 2)) return false;

   let start = gene.substring(0, 3);
   if (start !== START_CODON) return false;

   gene = gene.substring(3);
   for (let i = 0; i < sequence.length; i++) {
      let codons = MAPPINGS[sequence[i]];
      let codon = gene.substring(0, 3);
      if (codons.indexOf(codon) === -1) return false;
      gene = gene.substring(3);
   }

   let stop = gene;
   return STOP_CODONS.indexOf(stop) !== -1;
}
```