Jerry Cain
CS 106AX

Handout #9
October 18, 2023

Section Handout #3: Complex Data Structures

Section handout written by Jerry Cain.

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 Friday's introduction to arrays, implement the top-level function called **`split`**.

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

Problem 2: Strings, Arrays, and Disguised Algorithms

a) Suppose the **perplexity** function is defined as follows:

```
function perplexity(array) {
  let befuddled = array[0];
  let baffled = [befuddled];
  for (let i = 1; i < array.length; i++) {
     befuddled = Math.max(array[i], befuddled + array[i]);
     baffled.push(Math.max(befuddled, baffled[i - 1]));
  }
  return baffled;
}
```

A quick glance of the code suggests the above function considers every element of the incoming array and returns a second array whose length is equal to that of the incoming one. Save for the 0th element, the **for** loop visits every element in **array** and ensures that some new element is introduced to the one that's ultimately returned.

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

Consider the execution of the above call to **perplexity**, and complete the diagram below to indicate the integer values present in the array return value.

b) Suppose the **conundrum** function is defined as follows:

```
function conundrum(str) {
  let len = 0;
  let result = [0];
  for (let i = 1; i < str.length; i++) {
     if (str.charAt(i) === str.charAt(len)) {
        len++;
        result.push(len);
     } else if (len === 0) {
        result.push(0);
     } else {
        len = result[len - 1];
        i--;
     }
  }
  return result;
}
```

A quick glance of the code suggests the above function crawls over the supplied string and returns an array of numbers. Except for index 0, the **for** loop examines every character in **str** and builds up an integer array whose length ultimately matches that of **str** itself.

Consider the execution of a call to **`conundrum("AAACAAAAAC")`** and complete the diagram below to indicate the integer values present in the returned result.

Problem 3: 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 eventually appears in the sequence, we call the number Keith.

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 4: RNA, Codons, and Data Structures

Ribonucleic acid—more commonly known as **RNA**—is a biological polymer of nucleotides, where each nucleotide consists of a five-carbon sugar (specifically, a ribose), one or more phosphate groups, and a nitrogen-containing molecule that has the chemical properties of a base.

Each nitrogenous base can be one of four different molecules: **uracil**, **adenine**, **guanine**, and **cytosine**, generally abbreviated as U, A, G, and C, respectively. While RNA is really a polymer of alternating sugars and phosphate groups, where these nitrogenous bases extend off of the polymer, RNA is best viewed—at least for the purposes of this problem—as a sequence of U's, A's, G's, and C's, because that sequence dictates how RNA operates on behalf of the larger organism around it.

Proteins, like RNA strands, are also polymers, but they are polymers of molecules called **amino acids**. A protein's three-dimensional structure and its specific biological function—maybe it's an antibody, maybe it's an enzyme, maybe it's something else—are determined by its amino acid sequence. And while there are only four different nucleotides in RNA strands, there are 20 different amino acids, with names like **alanine**, **cysteine**, **glutamic acid**, and **tryptophan**.

Substrings of RNA called **genes** effectively instruct living cells how to **synthesize** proteins. Here's a simplified version of how RNA encodes protein structure:

- Each gene is taken to be a sequence of nucleotide triplets called **codons**, and each codon is taken as an instruction to include some amino acid in a larger protein. If two codons are side by side in a gene, then the amino acids they encode are side by side in the corresponding protein. So, a gene fragment like `"AGAUGGUGC"` is really a sequence of three codons, `"AGA"`, `"UGG"`, and `"UGC"`. `"AGA"` maps to arginine, `"UGG"` maps to tryptophan, and `"UGC"` maps to cysteine. If a gene's first nine nucleotides—restated, three codons—are `"AGAUGGUGC"`, then the encoded protein would begin with arginine, tryptophan, and cysteine.
- Because there are $4^3 = 64$ different codons and a mere 20 amino acids, many of the codons are treated as synonyms and encode the same amino acid.
- The beginning of every gene is marked by the **start codon**, which is always `"AUG"`. As it turns out, the amino acid methionine is also encoded by `"AUG"`, but that mapping is only active after we've seen `"AUG"` the first time.
- The end of every gene is marked by any one of three different **stop codons**: `"UAA"`, `"UGA"`, `"UAG"`.

The JavaScript map constant below contains 20 keys, each of which is the name of some amino acid. Each key is mapped to an array of all codons enconding it.

```
const START_CODON = "AUG";
const STOP_CODONS = ["UAA", "UGA", "UAG"];
const MAPPINGS = {
   "alanine": ["GCU", "GCC", "GCA", "GCG"],
   "arginine": ["CGT", "CGC", "CGA", "CGG", "AGA", "AGG"],
   "asparagine": ["AAU", "AAC"],
   "aspartic acid": ["GAU", "GAC"],
   "cysteine": ["UGU", "UGC"],
   "glutamine": ["CAA", "CAG"],
   "glutamic acid": ["GAA", "GAG"],
   "glycine": ["GGU", "GGC", "GGA", "GGG"],
   "histidine": ["CAU", "CAC"],
   "isoleucine": ["AUU", "AUC", "AUA"],
   "leucine": ["UUA", "UUG", "CUU", "CUC", "CUA", "CUG"],
   "lysine": ["AAA", "AAG"],
   "methionine": ["AUG"],
   "phenylalanine": ["UUU", "UUC"],
   "proline": ["CCU", "CCC", "CCA", "CCG"],
   "serine": ["UCU", "UCC", "UCA", "UCG", "AGU", "AGC"],
   "threonine": ["ACU", "ACC", "ACA", "ACG"],
   "tryptophan": ["UGG"],
   "tyrosine": ["UAU", "UAC"],
   "valine": ["GUU", "GUC", "GUA", "GUG"],
};
```

Using the next page, write a function called **`mappingIsValid`**, which accepts a gene and an array of amino acids, and returns **`true`** if and only if the provided gene begins with the start codon **`"AUG"`**, ends with one of the three stop codons **`"UAA"`**, **`"UGA"`**, or **`"UAG"`**, and everything in between precisely encodes the supplied sequence of amino acids and nothing else.

```
function mappingIsValid(gene, sequence) {
```