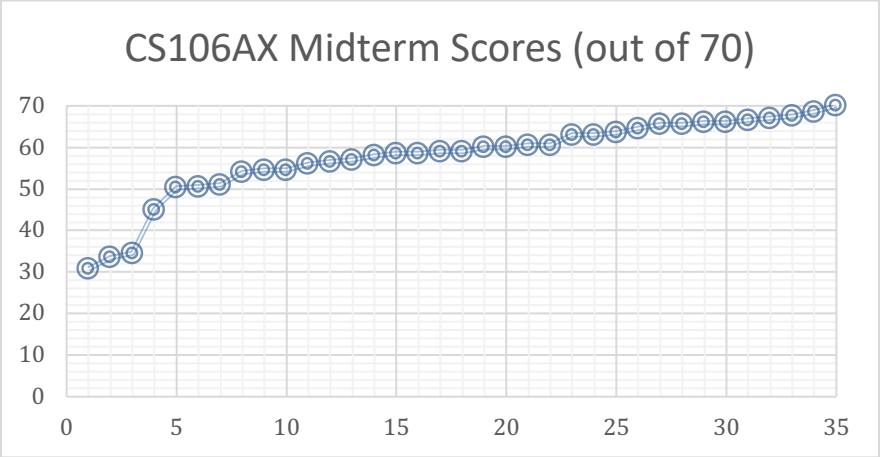
Jerry Cain
CS106AX

Handout #16S
November 6th, 2024

Midterm Examination Solution

Wonder CAs Ben and Rachel staff spent the weekend reading through your midterms, and the section leaders came through yesterday to help out grade the final problem. I'm happy to report the midterms have been fully graded, and your results have already been published via Gradescope at https://www.gradescope.com. The exam was intended to be challenging, but many of you did brilliantly, and as a group you exceeded expectations. I'm happy to go with my traditional curve for an accelerated course where I set the median grade to map to one of the higher A-'s.

The complete histogram of score is presented below, where each circle represents a single exam score out of 70 points:



You can determine your letter grade by looking up your score in the following table:

Median = 59.0

Range	Grade	N
67–70	A+	4
60–66.5	A	13
56–59.5	A–	8
52–55.5	B+	3
40–51.5	B	4
30–39.5	B–	3

Solution 1: Simple JavaScript expressions, statements, and methods (10 points)

(1a) [3 points] Compute the value of each of the following JavaScript expressions:

`4 % 3 - 15 % 1`	`1`
`10 === 10 && 29 / 0 === 29`	error, **`undefined`**, or **`false`**
`100000 + 3100 + "20" + 6 * 4`	`"1031002024"`

(1b) [3 points] Assume that the function **`ghost`** has been defined as given below:

```
function ghost(n) {
    let pumpkin = 0;
    while (n > 0) {
        let lantern = Math.floor(n / 10) * 10;
        for (let i = lantern; i < n; i++) {
            pumpkin++;
        }
        n = Math.floor(lantern / 10);
    }
    return pumpkin;
}
```

What is the value of **`ghost(102924)`**?

The **`ghost(102924)`** call returns **`18`**. The algorithm at play is a roundabout way of isolating each of the incoming parameter's digits and adding them together.

(1c) [4 points] What output is produced by a call to **`carrot()`**?

```
function carrot() {
   let broccoli = "squash";
   let corn = function(x, y, s) {
      return s.substring(x) + broccoli.substring(x, y);
   };
   broccoli = vegetable(corn, broccoli.indexOf("u"), broccoli.lastIndexOf("s"));
   console.log(broccoli);
}

function vegetable(fn, x, y) {
   let beet = fn(x, y, "cucumber");
   beet += "WXYZ".charCodeAt(3) - "ABCD".charCodeAt(0);
   return beet.substring(1, beet.length - 1);
}
```

A call to **`carrot()`** always prints **`umberua2`**.

Solution 2: Using graphics and animation (15 points)

Implement a graphical program that responds to an unbounded number of mouse clicks. Each mouse click should draw a small square, 40px by 40px with its center coinciding with the click coordinate. The fill color of each square should be randomly chosen to be either `"Green"` or `"Blue"`, each of the two choices equally likely and independent of previously drawn squares.

Once introduced, each square should fall toward—and eventually beyond—the bottom of the screen until it's no longer visible, at which point it should be removed from the graphics window. Green squares fall at a rate of two pixels every 30 milliseconds, whereas blue squares fall at a rate of just one pixel every 30 milliseconds.

Our own solution is presented below.

```
/*
 * Function: FallingSquares
 * ------------------------
 * Provided the entry point for the Falling Squares program, which
 * draws blur and green squares in response to user mouse clicks and
 * and then animates their constant-velocity descent toward the bottom
 * of the graphics window.
 */
function FallingSquares() {
   let gw = GWindow(WINDOW_WIDTH, WINDOW_HEIGHT);
   let onClickAction = function(e) {
         let color = randomChance() ? "Blue" : "Green";
         let square = createSquare(e.getX(), e.getY(),
                                   SQUARE_DIMENSION, color);
         gw.add(square);
         let deltay = color === "Green" ? 2 : 1
         let step = function() {
               square.move(0, deltay);
               if (square.getY() >= WINDOW_HEIGHT) {
                     gw.remove(square);
                     clearInterval(timer);
               }
         };
         let timer = setInterval(step, STEP_TIME);
   };

   gw.addEventListener("click", onClickAction);
}

function createSquare(cx, cy, side, color) {
   let square = GRect(cx - side/2, cy - side/2, side, side);
   square.setColor(color);
   square.setFilled(true);
   return square;
}
```

Solution 3: Strings (15 points)

In Assignment 4, rotor setting strings like `"AAA"`, `"JLY"`, and `"BZZ"` are used to represent the positions of the three rotors. Consecutive mouse events prompt the rotors to advance as a base-26 automobile odometer might. A mouse event would trigger an initial rotor setting of `"JLY"` to advance to `"JLZ"`, and a second mouse event would prompt that `"JLZ"` to advance to `"JMA"`. Note the second mouse event prompted the `"Z"` to wrap around to `"A"`, which triggered the neighboring rotor to advance one as well. And to be clear, advancing `"ZZZ"` rotates everything around to deliver `"AAA"`.

The idea of advancing a rotor setting needn't be confined to strings of length 3. Had the number of rotors been 8 instead of 3, we'd be speaking about rotor settings like `"BEESBUZZ"` advancing to `"BEESBVAA"` (note the double carry) and then `"BEESBVAB"`, and so on.

Here are some sample calls:

```
                advanceRotorSetting("JLY") ⇒ "JLZ"
                advanceRotorSetting("MZZ") ⇒ "NAA"
                advanceRotorSetting("ZZZ") ⇒ "AAA"
           advanceRotorSetting("BEESBUZZ") ⇒ "BEESBVAA"
  advanceRotorSetting("QZZZZZZZZZZZZZZZ") ⇒ "RAAAAAAAAAAAAAAA"
```

Our own solution is presented below.

```
/*
 * Function: advanceRotorSetting
 * -----------------------------
 * Accepts the supplied rotor setting and returns the
 * setting that would result from a single mousedown event.
 */
function advanceRotorSetting(setting) {
   let next = "";
   let carry = true;
   let base = "A".charCodeAt(0);
   for (let i = setting.length - 1; i >= 0; i--) {
      let ch = setting.charAt(i);
      if (carry) {
         let offset = ch.charCodeAt(0) - base;
         offset++;
         offset %= 26;
         ch = String.fromCharCode(base + offset);
         carry = offset === 0;
      }
      next = ch + next;
   }
   return next;
}
```

Solution 4: Arrays (15 points)

A ***polydivisible*** numbers are positive integers such that its first k digits, when taken as a number, are divisible by k, for all reasonable values of k. For example, 8,076 is polydivisible, because:

- 8 is divisible by 1
- 80 is divisible by 2
- 807 is divisible by 3
- 8,076 is divisible by 4

Other examples of polydivisibles? 30,080, 1,652,588, 444,402,009, 80,480,408,404, and 76,245,056,107,220 are all examples of polydivisible numbers. And as it turns out, the set of all polydivisible numbers is finite, and 3,608,528,850,368,400,786,036,725, at 25 digits, is the largest.

Here we describe an iterative algorithm that generates a sorted array of all polydivisible numbers. Our algorithm relies on the fact that that, say, a seven-digit number can only be polydivisible if its first six digits also comprise a polydivisible number. More generally, a k-digit number can only be polydivisible if its first k-1 digits comprise a polydivisible number as well.

Write a JavaScript function called **`generate`** that programmatically generates all of the polydivisible numbers. Rather than bundle all numbers in a single array, return an array of 26 lists: the 0th list is the empty list (since there are no polydivisible numbers with zero digits), the 1th list is `[1,2,3,4,5,6,7,8,9]` (since all single digit numbers are divisible by 1), the 2nd list contains all two-digit numbers divisible by 2, the 3rd list contains all three-digit polydivisible numbers, and so forth, up to a 25th list that contains 3,608,528,850,368,400,786,036,725 (which is the only such 25-digit polydivisible).

Your function should programmatically generate the entire list of 26 lists, where the list at index 1 is used to generate the list at index 2, which in turn is used to generate the list at index 3, and so forth. As you discover that, say, 807 is a polydivisible number, you'll use that 807 to generate 8070, 8071, 8072, etc., though 8079—just multiply 807 by 10 and add 0 or 1 or 2, etc.—and include those that are incidentally polydivisible as well.

Our implementation is presented on the next page.

```
/**
 * Function: generate
 * ------------------
 * Constructs and returns an array of all polydivisible numbers. The
 * polydivisible numbers are grouped by digit-count, so that the kth
 * array entry is itself an array of all k-digit polydivisible numbers.
 */
function generate() {
   let numbers = [];
   numbers.push([]);
   numbers.push([1, 2, 3, 4, 5, 6, 7, 8, 9]);
   length = 2;
   while (true) {
      let baselines = numbers[numbers.length - 1];
      let extensions = [];
      for (let i = 0; i < baselines.length; i++) {
         let baseline = baselines[i];
         for (let digit = 0; digit <= 9; digit++) {
            let candidate = 10 * baseline + digit;
            if (candidate % length === 0) extensions.push(candidate);
         }
      }
      if (extensions.length === 0) { return numbers; }
      numbers.push(extensions);
      length++;
   }
}
```

As it turns out, the largest polydivisible number is so large that JavaScript can't store it in a traditional integer variable. In order for the above implementation to truly work, you'd employ a **`BigInt`**, which you get by appending an **`n`** to the end of any integer constant involved in the construction of a polydivisible number. You didn't need to know this, of course.

```
function generate() {
   let numbers = [];
   numbers.push([]);
   numbers.push([1n, 2n, 3n, 4n, 5n, 6n, 7n, 8n, 9n]);
   length = 2n;
   while (true) {
      let baselines = numbers[numbers.length - 1];
      let extensions = [];
      for (let i = 0; i < baselines.length; i++) {
         let baseline = baselines[i];
         for (let digit = 0n; digit <= 9n; digit++) {
            let candidate = 10n * baseline + digit;
            if (candidate % length === 0n) extensions.push(candidate);
         }
      }
      if (extensions.length === 0) { return numbers; }
      numbers.push(extensions);
      length++;
   }
}
```

Solution 5: Working with data structures (15 points)

By now, you're all quite familiar with the game of Wordle, even if you'd somehow missed news of it prior to CS106AX's Assignment 3. Of course, the goal is to uncover a secret word via a series of six or fewer educated guesses.

Assume the following data structure has been built up to store information on behalf of the many people who played Wordle in any given day.

```
let database = {
   secret: "BERET",
   guesses: [
         ["SLATE", "THERE", "BERET"],
         ["ADIEU", "STORE", "BERET"],
         ["HEART", "BERET"],
         ["PLUMB", "STORE", "BRASH", "BERET"],
         ["PLUMB", "BRISK", "BREAK", "REBEC", "BETEL", "BESET"],
         ["BERET"],
         // many, many more guess sequences
         ["SILLY", "CRANE", "BERTH", "BEVEL", "BERET"],
         ["PLATE"]
      ]
};
```

The **`database`** variable is a JavaScript aggregate with just two fields, **`secret`** and **`guesses`**. The first maps to the secret word of the day, and the second maps to a nonempty array of subarrays, where each subarray tracks the nonempty sequence of words a user entered while playing before either winning, losing, or giving up. You may assume all subarrays consist of only five-capital-letter words, though you shouldn't assume all subarrays end with a match, since that's clearly not the case with the fifth and the last of the subarrays.

Implement **`mostPopularStartWord`**, which accepts a data structure like that above and returns the start word that most often led to victory. Your implementation should be sure to ignore the subarrays that don't end in a match, and if two or more words are tied for most popular, you can return any one of them.

Our implementation is presented on the next page.

```
/**
 * Function: mostPopularStartWord
 * ------------------------------
 * Traverses the database of information about how everyone did on
 * today's Wordle and returns the most common start word used by those
 * who completed the puzzle.
 */
function mostPopularStartWord(database) {
   let map = {}, secret = database.secret, guesses = database.guesses;
   for (let i = 0; i < guesses.length; i++) {
      if (guesses[i][guesses[i].length - 1] === secret) {
         if (map[guesses[i][0]] === undefined) {
            map[guesses[i][0]] = 0;
         }
         map[guesses[i][0]]++;
      }
   }

   let winner = "", count = 0;
   for (let key in map) {
      if (map[key] > count) {
         winner = key;
         count = map[key]
      }
   }

   return winner;
}
```