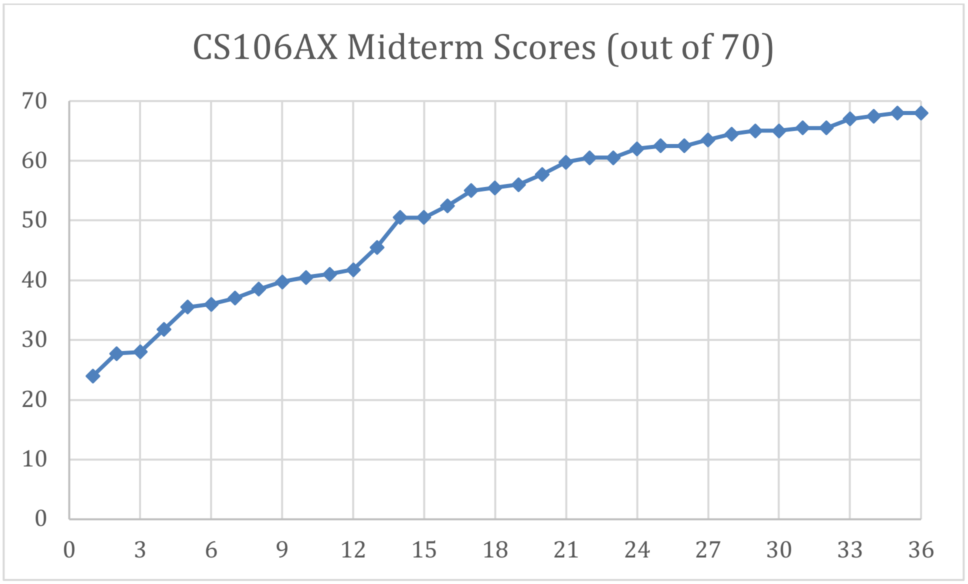
Jerry Cain
CS106AX

Handout #15S
October 31st, 2025

Midterm Examination Solution

Wonder CA Ben and section leaders spent their Sunday afternoon grading all of the midterms. I'm happy to report that they've been fully graded, and your results have already been published via Gradescope at https://www.gradescope.com. The exam was challenging, but many of you still 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 the A/A-.

The complete histogram is presented below, where each diamond 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 = 55.75

Range	Grade	N
65–70	A+	8
56–64.5	A	10
50–55.5	A–	5
45–49.5	B+	1
35–44.5	B	8
24–34.5	B–	4

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

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

`8 / 2 + 70 % 40`	**34**
`5 < 3 && 10 / 0 === 0`	**false**
`10 * 10 + "DOE" + 7 + 8`	**"100DOE78"**

(1b) [3 points] Suppose that the function **`accumulation`** is defined as follows:

```
function bumfuzzle() {
   let cabotage = [];
   for (let i = 0; i <= 12; i++) {
      cabotage.push(0);
      for (let j = i; j > 0; j--) {
         cabotage[j] += j;
      }
   }
   return cabotage;
}
```

If you look at the code, you'll see that the function pushes a new value onto the array for every value between 0 and 12, which means that the array will eventually contain 13 elements with indices ranging from 0 to 12. Work through the function carefully and indicate the value of each of the elements in the array that **`bumfuzzle`** returns:

0	12	22	30	36	40	42	42	40	36	30	22	12
0	1	2	3	4	5	6	7	8	9	10	11	12

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

```
function world() {
   let japan = "Namibia";
   let france = function(x, y, s) {
      return japan.substring(x, y) + s.substring(y);
   };
   let peru = alliance(france, japan.indexOf("m"), japan.lastIndexOf("i"));
   console.log(peru);
}

function alliance(f, x, y) {
   let fiji = f(x, y, "Mauritius");
   fiji += "Kazakhstan".charCodeAt(0) - "Jordan".charCodeAt(0);
   return fiji;
}
```

`mibtius1`

Solution 2: Using graphics and animation (15 points)

Implement a graphical program that introduces a single, randomly placed, randomly colored circle into the graphics window every 1000 milliseconds. Each circle, independently of any others, jiggles about by moving 3 pixels in a randomly chosen direction every 80 milliseconds. Even though circles may drift outside the graphics window, you needn't worry about removing them. Let them continue to jiggle out of view and possibly drift back in. Don't worry if the initial placement of the circle overlaps the edge of the graphics window. Just make sure its center is on the screen.

When the user clicks one of the circles, a new circle with precisely the same color and location as the one clicked is added to the graphics window. From that point one, the two circles jiggle about independently and can eventually drift very far apart from one another. All circles respond to mouse clicks, so that circles created by mouse clicked can themselves be clicked and be cloned. Clicks that miss all circles have no effect. And if there are multiple circles overlaying a particular click location, your click should split just the topmost one.

The program is designed to playfully imitate **mitosis**, which is fancy speak for cell division. The graphics window is a petri dish, the circles are cells, the jiggling is gentle Brownian motion, and mouse clicks prompt a single cell to split into two copies of itself.

A few things:

- You already know that all objects respond `setFillColor`, which allows you to specify what color the interior of a filled object should be. There is also the `getFillColor` method, that returns that color as a string.
- You've seen the `move` method a few times, but you may not have seen the `movePolar` method. `movePolar` takes a distance in pixels and an angle in degrees. So, a call to `obj.movePolar(10, 0)` moves `obj` 10 pixels to the right, `obj.movePolar(10, 270)` moves `obj` down, and `obj.movePolar(10, 135)` moves `obj` an equal number of pixels up and to the left. (All integer angles between 0 and 359, inclusive, are equally likely to be chosen with each jiggle.)
- The are two types of animation—one that creates new cells every second, and one that jiggles a cell. All animations should be implemented to run forever, so that you never need to call `clearInterval`.
- Remember the random library entries: `randomInteger`, `randomReal`, `randomChance`, and `randomColor`. You'll use a subset of them when writing your answer.

Use the space on the next page to write out your implementation.

(space for the answer to problem #2 appears on the next page)

```
/* Constants (in pixels) */
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;
const CELL_RADIUS = 10;
const CELL_DIAMETER = 2 * CELL_RADIUS;
const DRIFT = 3;

/* Constants (in milliseconds) */
const BIRTH_STEP = 1000;
const JIGGLE_STEP = 80;

/* Main program */
function Mitosis() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   let addCell = function(cx, cy, color) {
      let cell = GOval(cx - CELL_RADIUS, cy - CELL_RADIUS,
                       CELL_DIAMETER, CELL_DIAMETER);
      cell.setFilled(true);
      cell.setFillColor(color);
      setInterval(function() {
         cell.movePolar(DRIFT, randomInteger(1, 360));
      }, JIGGLE_STEP);
      gw.add(cell);
   };

   let step = function() {
      let cx = randomInteger(CELL_RADIUS, GWINDOW_WIDTH - CELL_RADIUS);
      let cy = randomInteger(CELL_RADIUS, GWINDOW_HEIGHT - CELL_RADIUS);
      addCell(cx, cy, randomColor());
   };

   let clickAction = function(e) {
      let circle = gw.getElementAt(e.getX(), e.getY());
      if (circle === null) return;
      addCell(circle.getX() + CELL_RADIUS,
              circle.getY() + CELL_RADIUS, circle.getFillColor());
   }

   setInterval(step, BIRTH_STEP);
   gw.addEventListener("click", clickAction);
}
```

Solution 3: Strings (15 points)

Some encryption schemes rely on ever-evolving encryption keys to more effectively disguise common letters like `'e'`, `'t'`, and `'a'`. For this problem, you'll write a function that encrypts a lowercase string using a **move-to-front encryption key transformation**.

The key is initially set to be the lowercase alphabet, `"abcdefghijklmnopqrstuvwxyz"`. For each character `ch` in the original word, the encryption scheme:

- finds the index of `ch` within the key,
- appends the character at that index within the alphabet to a running encryption, and
- moves `ch` to the front of the key while keeping the other letters in the same order.

As an example, the word `"ooze"` would be transformed into `"oazg"`, and here's why:

- Start with the key of `"abcdefghijklmnopqrstuvwxyz"`.
- The first letter is `'o'` and appears at index 14 of the key.
    - Append the 14th letter of the alphabet—that is, an `'o'`—to the encryption.
    - Move `'o'` to the front of the key, yielding `"oabcdefghijklmnpqrstuvwxyz"`.
- The second letter is another `'o'`, but it now appears at index 0 of the key.
    - Append the 0th letter of the alphabet, `'a'`, to the encryption.
    - The key remains the same.
- The third letter `'z'` appears at index 25 of the key.
    - Append the 25th letter of the alphabet, `'z'`, to the encryption.
    - Move `'z'` to the front of the key to yield `"zoabcdefghijklmnpqrstuvwxy"`.
- The final letter `'e'` appears at index 6 of the key.
    - Append the 6th letter of the alphabet, `'g'`, to the encryption.
    - Move `'e'` to the front of the key, producing `"ezoabcdfghijklmnpqrstuvwxy"`.

The final encryption? `"oazg"`! That's what would be returned.

Use the next page to implement the `encrypt` function to encrypt the supplied word—assumed to consist of lowercase letters and nothing else—and returns its encryption.

```
const ALPHABET = "abcdefghijklmnopqrstuvwxyz";
function encrypt(word) {
   let key = ALPHABET;
   let encryption = "";
   for (let i = 0; i < word.length; i++) {
      let index = key.indexOf(word[i]);
      let ch = ALPHABET.charAt(index);
      encryption += ch;
      key = ch + key.substring(0, index) + key.substring(index + 1);
   }
   return encryption;
}
```

Solution 4: Arrays (15 points)

The city of San Francisco relies on **ranked choice voting**—also known as instant runoff voting—for its mayoral elections. Rather than voting for a single candidate, those casting ballots vote for up to **three** candidates, ranking them 1st, 2nd, and 3rd.

Each ballot is a short array of at most three strings, and the accumulation of all ballots is represented as an array. The first five ballots of what in practice would be hundreds of thousands in a real San Francisco mayoral election might look like this:

```
ballots = [["Breed", "Lurie"], ["Peskin", "Farrell", "Lurie"],
           ["Farrell"], ["Safai", "Lurie", "Breed"], ["Farrell", Safai"],
           // many, many other ballots ];
```

Initially, only true first-choice votes matter, and if a single candidate gets the majority of all first-choice votes, then that candidate wins. Seems right.

In a crowded field of candidates, however, that never happens. There were, for example, 17 official candidates in San Francisco's mayoral election last November, and Daniel Lurie, who eventually won over incumbent London Breed, got only 26% of the first-choice votes. In that case, the candidate with the smallest number of rank-one votes is eliminated and removed from all ballots everywhere, promoting all second- and third-choice votes to first- and second-choice votes to close any gaps.

If, for example, a tally confirms that Mark Farrell received the smallest number of first-choice votes, the ballots would be updated to look like this:

```
ballots = [["Breed", "Lurie"], ["Peskin", "Lurie"],
           ["Safai", "Lurie", "Breed"], ["Safai"],
           // many, many other ballots ];
```

The first and fourth ballots are left alone, but all others are updated to reflect Farrell's disappearance. And that one ballot with a standalone vote for Farrell is removed in its entirety, since it becomes an empty ballot after he's eliminated. The other two ballots would see Daniel Lurie promoted from third to second place and Ahsha Safai lifted from second to first.

The process is repeated until it leaves just one candidate with a majority. In fact, last November, this very process was applied 14 times before Daniel Lurie prevailed with 53% of all remaining first-choice votes.

Implement the `eliminate` function that, given an array of `ballots`, identifies the candidate with the least number of first-choice votes, and returns a **new** array of ballots where all traces of that candidate have been removed (and all empty ballots have been removed as well). Note the incoming `ballots` array itself should **not** be modified. For simplicity, assume there are at least three candidates, all candidates have at least one rank-one vote, that no single candidate has a clear majority, and that no one ever has the same number of first-choice votes.

(space for the answer to problem #4 appears on the next page)

```
function eliminate(ballots) {
   let counts = {}
   for (let i = 0; i < ballots.length; i++) {
      let top = ballots[i][0];
      if (counts[top] === undefined) counts[top] = 0;
      counts[top]++;
   }

   let goner = null;
   let least = ballots.length + 1;
   for (let candidate in counts) {
      if (counts[candidate] < least) {
         goner = candidate;
         least = counts[candidate];
      }
   }

   let revised = [];
   for (let i = 0; i < ballots.length; i++) {
      let ballot = ballots[i];
      let updated = []
      for (let j = 0; j < ballot.length; j++) {
         let candidate = ballot[j];
         if (candidate != goner) updated.push(candidate);
      }
      if (updated.length > 0) revised.push(updated);
   }

   return revised;
}
```

Problem 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, five-letter English word via a series of six or fewer guesses. For each guess, the game identifies **correctly placed** letters by shading them green and **correctly guessed but incorrectly placed** letters by shading them yellow.

A JavaScript object modeling a single game of Wordle might look like this:

```
let play = {
   secret: "fetid",
   guesses: [
      { guess: "plate", green: [], yellow: [3, 4], nowhere: ["p", "l", "a"] },
      { guess: "tenor", green: [1], yellow: [0], nowhere: ["n", "o", "r"] },
      { guess: "feist", green: [0, 1], yellow: [2, 4], nowhere: ["s"] },
      { guess: "fetid", green: [0, 1, 2, 3, 4], yellow: [], nowhere: [] }
   ]
};
```

The above includes the `secret` word and the series of `guesses` that led to a win. The sequence of guesses is itself modeled as an array where the first guess occupies position 0, the second guess occupies position 1, and so forth. Each guess is represented as a smaller object with precisely four keys: the `guess`, the indices of the correctly placed `green` letters, the indices of the correctly guessed, incorrectly placed, `yellow` letters, and an array of letters within the guess that are truly `nowhere` to be found in the secret word.

Some versions of Wordle require that green letters stay put in subsequent guesses and that letters identified as irrelevant never appear again. (For this problem, we'll ignore yellow letters, since the information they provide is too much to manage in a timed exam.)

As it turns out, the game depicted by `play` respects both requirements.

- The player's second guess confirmed that an `'e'` occupies index 1, so every guess thereafter included an `'e'` at the same position. When the player learned the secret word began with `'f'`, she made sure the next guess began with an `'f'`, too.
- When the player's first guess made it clear that `'p'`, `'l'`, and `'a'` had no place in the secret word, she was careful to exclude those three letters from all guesses thereafter. She learned from the second guess that `'n'`, `'o'`, and `'r'` were no good, and from the third guess that `'s'` shouldn't be used again either.

This is an example of what we'll call a **perfect play**. If, however, the player's third guess had been `yeast` instead of `feist`, that would have been a mistake, since `a` was identified as forbidden by the initial guess.

For this problem, you're to implement a function called `playedPerfectly`, which accepts a game object like the one presented above and returns `true` if and only if the game was played perfectly, and `false` otherwise.

(space for the answer to problem #5 appears on the next page)

```
function playedPerfectly(game) {
   let guesses = game.guesses;
   for (let i = 0; i < guesses.length - 1; i++) {
      for (let j = 0; j < guesses[i].green.length; j++) {
         if (guesses[i + 1].green.indexOf(guesses[i].green[j]) === -1)
            return false; // i + 1th guess missing a green location
      }
   }

   let outlawed = {}; // could also use an array, but maps are cool
   for (let i = 0; i < guesses.length; i++) {
      for (let j = 0; j < guesses[i].nowhere.length; j++) {
         let forbidden = guesses[i].nowhere[j];
         if (outlawed[forbidden] !== undefined) return false;
         outlawed[forbidden] = true;
      }
   }

   return true;
}
```