CS 106AX Midterm Solution

The course staff spent several hours grading your midterms over the weekend, so I’m happy to report they’ve been graded, and your graded midterms will be published via Gradescope ahead of today’s lecture. The exam was intended to be challenging, but many of you did brilliantly, and most of you did well beyond what I expected I’m happy to go with a traditional curve for an accelerated course, where I set the median grade to sit just above the A-/B+ border.

The complete histogram of grades is presented below, where each dot represents a single exam score (technically out of 70 points, with space for extra credit of up to 4 points).

![Histogram of grades](image)

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

<table>
<thead>
<tr>
<th>Range</th>
<th>Grade</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>71–74</td>
<td>A+</td>
<td>4</td>
</tr>
<tr>
<td>65–70</td>
<td>A</td>
<td>13</td>
</tr>
<tr>
<td>60–64</td>
<td>A−</td>
<td>15</td>
</tr>
<tr>
<td>55–59</td>
<td>B+</td>
<td>8</td>
</tr>
<tr>
<td>47–54</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>40–46</td>
<td>B−</td>
<td>2</td>
</tr>
<tr>
<td>32–39</td>
<td>C+</td>
<td>1</td>
</tr>
<tr>
<td>21–31</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>13–20</td>
<td>C−</td>
<td>2</td>
</tr>
<tr>
<td>00–12</td>
<td>D</td>
<td>0</td>
</tr>
</tbody>
</table>

Median = 62 (85.7%)
Solution 1: Simple JavaScript expressions and methods [10 points/70 total]

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 + 2 \times 3 % 5$</td>
<td>2</td>
</tr>
<tr>
<td>&quot;C&quot; === &quot;CC&quot;</td>
<td></td>
</tr>
<tr>
<td>$106 + &quot;AX&quot; + 1 + 2 + 3$</td>
<td>106AX123</td>
</tr>
</tbody>
</table>

(1b)  [3 points] Assume that the method `halloween` has been defined as follows:

```javascript
function halloween(word) {
    let result = "";
    while (word.length > 1) {
        if (word.charAt(0) > word.charAt(1)) {
            word = word.substring(0, Math.floor(word.length/2));
        } else {
            word = word.substring(Math.floor(word.length/2));
        }
        result += word;
    }
    return result;
}
```

What is the value returned by `halloween("hocuspocus")`?

Answer to problem 1b:  pocuspop
(1c) [4 points] What output is printed by the following Problem1c program?

```
function Problem1c() {
    let doris = "bostonterrier";
    let boston = function(s, y) {
        return doris.substring(0, y) + s.substring(y);
    };
    let terrier = function(s, x) {
        let toy = boston(s, x);
        toy += String.fromCharCode("4".charCodeAt(0) + 5);
        return toy;
    };
    doris = dogpark(boston, terrier);
    doris.concat("play");
    console.log(doris);
}

function dogpark(f, g) {
    return f("livertreat", 6) + " " + g("puppyfood", 6);
}
```

Answer to problem 1c:

```
bostonreat bostonood9
```
Solution 2: Graphics, Callbacks, & Animation [15 points/70 total]

In this problem, you’ll be implementing the controls for a miniature game of Snake, called “ByteSnake”!

You will start with a small square, as below. You’ll animate it so that every TIME_STEP, it moves by a pre-determined amount; and then, you’ll allow the player to change the direction it moves. We’ll build up the solution step by step!

You can control the snake in the demos using the W (to go up), A (to go left), S (to go down), and D (to go right) keys.

Part 1: Animating the Player
We’ve created the player’s cube for you– your job is to make it move! We’ll start by having it always move to the right– marching infinitely past the edges of the screen. We expect you to write about 6 lines of code for this problem. You'll find the SPEED constant helpful.

Part 2: Controlling the Player
Now that the player is animated, let’s create the ability to control it! We expect you to write 20 or so lines of code for this problem.

For this problem, you'll be using the keydown event, just like in Wordle. We provide the same getKeystrokeLetter you used on that assignment.

You will need to create an event listener for the keydown event. When the event fires, you should use the getKeystrokeLetter function to get which key was pressed.

- For W, the snake should move up
- For A, the snake should move left
- For S, the snake should move down
- For D, the snake should move right
  - In any other case, simply ignore the keystroke.

Once you figure out what direction you want the snake to be moving, you should change the X and Y velocities to move the snake in that direction.

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

```javascript
function animatePlayer(gw) {
  // This code is given, and correct
  let player = GRect(GWINDOW_WIDTH / 2 - PLAYER_SIZE,
                     GWINDOW_HEIGHT / 2 - PLAYER_SIZE, PLAYER_SIZE, PLAYER_SIZE);
  gw.add(player);

  //**************************
  *     PART 1      *
  //**************************

  // Step 1. Create variables to keep track of the X and Y velocities. (We expect 2 lines of code.)
  let vx = SPEED;
  let vy = 0;

  // Step 2. Every TIME_STEP, your player should move by the current X and Y velocity.
  setInterval(step, TIME_STEP);

  function step() {
    player.move(vx, vy);
  }

  //**************************
  *     PART 2      *
  //**************************

  // Step 1. Create your keydown callback function.
  function onKeydown(e) {
    let letter = getKeystrokeLetter(e);

    if (letter === "w") {
      vx = 0;
      vy = -SPEED;
    } else if (letter === "a") {
      vx = -SPEED;
      vy = 0;
    } else if (letter === "s") {
      vx = 0;
      vy = SPEED;
    } else if (letter === "d") {
      vx = SPEED;
      vy = 0;
    }
  }

  // Step 2. Attach your callback function to gw as event listeners for the keydown event.
  gw.addEventListener("keydown", onKeydown);
}
```

Part 3: Wrap-around (EXTRA CREDIT)

Now that you’ve implemented player control, the last thing to do is make it so that the
player wraps around, rather than falling off the sides! We expect you to write about 15 lines of code for this part.

**This is an extra credit problem. Come back to it at the end if you have time to complete it.**

For this problem, you’ll be modifying the function you created in part 1. You may assume that `PLAYER_SIZE`, `GWINDOW_HEIGHT`, `GWINDOW_WIDTH`, and `SPEED` are all set so you’ll never have to worry about the player being halfway offscreen in any direction.

The player should wrap around on all sides of the GWindow; if they are past the bottom, they should wrap around to the top, and vice versa; and if they are past the right of the screen, they should wrap around to the left, and vice versa.

```javascript
// The step function above changes to read as so:
function step() {
    player.move(vx, vy);

    //**********************************
    // PART 3 - EXTRA CREDIT
    //**********************************
    if (player.getX() >= GWINDOW_WIDTH) {
        player.setLocation(0, player.getY());
    } else if (player.getX() <= -PLAYER_SIZE) {
        player.setLocation(GWINDOW_WIDTH - PLAYER_SIZE, player.getY());
    } else if (player.getY() >= GWINDOW_HEIGHT) {
        player.setLocation(player.getX(), 0);
    } else if (player.getY() <= -PLAYER_SIZE) {
        player.setLocation(player.getX(), GWINDOW_HEIGHT - PLAYER_SIZE);
    }
}
```
Solution 3: Strings [15 points/70 total]

For this problem, you’ll leverage your understanding of strings to generate a random reflector that you might use in your implementation of Assignment 4’s Enigma.

Recall that the reflector encrypts each letter of the alphabet to some other letter, and that all reflector encryptions are reversible. Thus, if "A" is encoded as "P", then "P" must be encoded as an "A".

The permutation for the reflector used in the Assignment 4 specification was:

"IXUFEZDAOMTKQJWNSRLCPBG"

Note that "A" is encoded as "I", "B" is encoded as "X", "C" is encoded as "U", and so forth. And because the permutation is a reflector, it must be the case that "I" maps back to "A", "X" maps back to "B", "U" maps back to "C", and so on. No character in the above permutation occupies its normal spot in the alphabet, since that would imply that character is its own encryption, and that’s not permitted.

For this problem, your job is to implement the buildReflector function, which constructs and returns a valid reflector permutation. Because this function is algorithmically complex, we give you the general algorithm in pseudocode:

```javascript
const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const STARTER = "--------------------------";

function buildReflector() {
    let reflector = STARTER;
    let candidates = ALPHABET;
    while (candidates.length > 0) {
        select a random position from the string of candidates, and remember the character there
        remove the selected character from the string of candidates
        select a random position from the updated string, and remember the character there
        remove the selected character from the string of candidates
        update the reflector so that each of the two selected characters occupies the other’s
        spot in the alphabet
    }
    return reflector;
}
```

Note that two characters are deleted from candidates on each iteration, which means that the body of the loop will always execute 13 times. And recall the randomInteger function from Assignment 1, where a call to randomInteger(a, b) for integers a and b is equally likely to return any integer between a and b, inclusive.

(space for the answer to problem #3 appears on the next page)
Answer to problem #3:

```javascript
const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const STARTER = "-------------------------";

function buildReflector() {
    let reflector = STARTER;
    let candidates = ALPHABET;
    while (candidates.length > 0) {
        let pos1 = randomInteger(0, candidates.length - 1);
        let ch1 = candidates.charAt(pos1);
        candidates =
            candidates.substring(0, pos1) + candidates.substring(pos1 + 1);
        let pos2 = randomInteger(0, candidates.length - 1);
        let ch2 = candidates.charAt(pos2);
        candidates =
            candidates.substring(0, pos2) + candidates.substring(pos2 + 1);
        pos1 = ch2.charCodeAt(0) - "A".charCodeAt(0);
        pos2 = ch1.charCodeAt(0) - "A".charCodeAt(0);
        reflector =
            reflector.substring(0, pos1) + ch1 + reflector.substring(pos1 + 1);
        reflector =
            reflector.substring(0, pos2) + ch2 + reflector.substring(pos2 + 1);
    }
    return reflector;
}
```
**Solution 4: Arrays [15 points/70 total]**

Consider the following permutation:

<table>
<thead>
<tr>
<th>6</th>
<th>3</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>0</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

The above sequence is a permutation of (that is, an arbitrary reordering of) the numbers 0 to 8, inclusive. What’s interesting is that we can identify **cycles** in the above permutation (and indeed, any permutation), by following the numbers around the array until we reach our starting point again. For example, the permutation above has 5 cycles as follows:

6 → 0, 3 → 5 → 1, 8 → 2, 4, and 7

The 3 → 5 → 1 cycle is implied by the permutation, because the 5 is at index 3 and therefore follows the 3, the 1 is at index 5 and therefore follows the 5, and the 3 is at index 1 and therefore follows the 1. Specifically, every number \( k \) in the permutation is part of exactly one cycle, and \( k \)’s successor in the cycle is at index \( k \).

Note that 8 and 2 are mutual successors of one another, since the 2 is at index 8 and the 8 is at index 2. That means 2 follows 8 follows 2 in another cycle separate from 3 → 5 → 1. And 4 is in its own cycle since it resides at index 4 and is its own successor. Cycles of size 1 are completely legit.

Each cycle can be represented as an integer array, and the collection of cycles for any given permutation can be expressed as an array of integer arrays. Hence, given a permutation

\[
[6, 3, 8, 5, 4, 1, 0, 7, 2]
\]

we can express that permutation’s partition into cycles as

\[
[[6, 0], [3, 5, 1], [8, 2], [4], [7]]
\]

(Note that all rotations of a cycle are equivalent, so that \([3, 5, 1]\) could have been \([5, 1, 3]\) or \([1, 3, 5]\) instead and all would have been good.)

Implement the **permutationToCycles** function, which accepts a valid permutation and returns a partition of that permutation as an array of integer arrays. You can trust that the input is a valid permutation, though you shouldn’t make any assumptions about the permutation’s length, as it could be a permutation of the numbers 0 through 8 as above, or it could be a permutation of the numbers 0 through 45, or 0 through 1234.

Implementation hint: For any permutation of size \( n \), maintain an array of \( n \) Booleans to track whether a particular integer in the permutation has already been processed.

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

```javascript
/*
* Function: permutationsToCycles
* -----------------------------
* Accepts a valid permutations on the first n natural numbers and returns the collection of cycles implied by it, as per the problem statement.
* 
* permutationsToCycles([6, 3, 8, 5, 4, 1, 0, 7, 2])
* -> [[6, 0], [3, 5, 1], [8, 2], [4], [7]]
* permutationsToCycles([0, 2, 3, 1, 5, 4])
* -> [[0], [2, 3, 1], [5, 4]]
* permutationsToCycles([7, 0, 1, 2, 3, 4, 5, 6])
* -> [[7, 6, 5, 4, 3, 2, 1, 0]]
*/

function permutationsToCycles(perm) {
  let used = [];
  for (let i = 0; i < perm.length; i++) used.push(false);
  let cycles = [];
  for (let pos = 0; pos < perm.length; pos++) {
    if (!used[pos]) {
      let cycle = [];
      let curr = pos;
      while (!used[curr]) {
        cycle.push(perm[curr]);
        used[curr] = true;
        curr = perm[curr];
      }
      cycles.push(cycle);
    }
  }
  return cycles;
}
```
Solution 5: Working with data structures [15 points / 70 total]

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 educated guesses. With each guess, the game identifies correctly placed letters by shading them green and further identifies correctly guessed but incorrectly placed letters by shading them yellow.

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

```javascript
let game1 = {
    secret: "rerun",
    guesses: [
        { guess: "reach",
          green: [0, 1],
          yellow: []
        },
        { guess: "refer",
          green: [0, 1],
          yellow: [4]
        },
        { guess: "rears",
          green: [0, 1],
          yellow: [3]
        },
        { guess: "rerun",
          green: [0, 1, 2, 3, 4],
          yellow: []
        }
    ]
};
```

The above includes the secret word and the series of guesses that led to the win. The sequence of guesses is itself modeled as an array—keyed by guesses—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 three keys: the guess, the indices of the correctly placed green letters, and the indices of the correctly guessed, incorrectly placed, yellow letters.

Some versions of Wordle require that a correctly placed letter never be moved in subsequent guesses. As it turns out, the game depicted by game1 respects this requirement: The leading "re" appears in every guess until the game is over. Restated, all letters, once shaded green, remain green for the lifetime of the game.

This is an example of what we’ll call perfect play.
Contrast the above to the game modeled here:

```javascript
let game2 = {
  secret: "tower",
  guesses: [
    {
      guess: "mouth",
      green: [1],
      yellow: [3]
    },
    {
      guess: "torch",
      green: [0, 1],
      yellow: [2]
    },
    {
      guess: "toner",
      green: [0, 1, 3, 4],
      yellow: []
    },
    {
      guess: "tours",
      green: [0, 1],
      yellow: [3]
    },
    {
      guess: "tower",
      green: [0, 1, 2, 3, 4],
      yellow: [2, 4]
    }
  ]
};
```

Here, the player seemingly goofed when guessing "tours". This mishap is obvious from the green properties in the objects modeling the two relevant guesses—the 3 and 4 that appear in `game2.guesses[2].green` are missing from `game2.guesses[3].green`.

For this problem, you’re to implement a function called `playedPerfectly`, which accepts a game object like those structured above and returns `true` if the game was played perfectly, and `false` otherwise. (You needn’t involve the yellow fields here. They’re simply included because it would have been strange to omit them from the discussion. A more sophisticated definition of perfect play might involve the yellow arrays, but in the interest of time, we won’t be that sophisticated.)

Your `playedPerfectly` implementation should examine the green arrays and confirm that once an index appears in some green array, it must appear in all subsequent ones. If your implementation detects a violation, it should return `false` without continuing. If no violations are discovered anywhere, you should return `true`.

Place your implementation on the next page.
function playedPerfectly(game) {
  for (let i = 0; i < game.guesses.length - 1; i++) {
    for (let j = 0; j < game.guesses[i].green.length; j++) {
      let pos = game.guesses[i].green[j];
      if (game.guesses[i + 1].green.indexOf(pos) === -1) {
        return false;
      }
    }
  }
  return true;
}