CS106AX: Programming Methodologies in JavaScript and Python

Autumn 2025 🍂, October 26, 2025

Problems by current staff, previous CS 106AX and CS 106A staff

CS106AX Midterm Review Session: Problems Dégustation

We won't go over all problems on the whiteboard, so the rest can be additional practice! :)

🥗 *Evaluation Appetizers*

1) `2024 + 3 % 2 * 2 - 2020` ________

2) `2025 > 2020 + 6 || 3 % 3 === 3` ________

3) `2 + 25 % (4 + 1) * 10 + 3` ________

4) `"BTS" <= "EXO" && "akb48" === "AKB48"` ________

Please note this question is not asking whether you prefer BTS or EXO's music more – if you listen to K-pop bands, that is 🎶.

5) `"BTS" <= "EXO" || 10 / 0 > 3.14` ________

6) `"messier" + 7 + 80` ________

7) `7 + 8 + "messier"` ________

8) `"NGC" + 44 + 86 + "messier" + 3 * 29` ________

🍵 *So Much Tea, Lets Backtrace*

1) **Parameter Passing, Return Value:** What two lines does a call to **`BensLife()`** print?

```
function BensLife(){
    let bank = {savings: 0, checking: 1000};
    let MONTH_RENT = 2000;
    console.log(payRent(bank, MONTH_RENT));
    console.log(MONTH_RENT);
}

function payRent(bank, rent){
    bank.checking -= rent;
    rent *= 2;
    let needSavings = bank.checking <= 0;
    return needSavings;
}
```

2) **Loop Trace Problem:** Assume that the function **`halloween`** 🎃 has been defined as follows:

```
function halloween(ghost1, ghost2){
     let treat = 0;
     let trick = 1;
     while (ghost1 > 0){
          let candy = ghost1 % 10 + ghost2 % 10;
          for (let i = 0; i < trick; i++){
               treat += candy;
          }
          trick *= 10;
          candy = Math.floor(ghost1 / 10);
          ghost1 = Math.floor(ghost2 / 10);
          ghost2 = candy;
     }
     return treat;
}
```

What is the value of **`halloween(2048, 4096)`**? __________

3) 🪢 **Silly Strings**

```
function sillyString(string){
  let spray = string.charAt(0);
  let rainbow = string.substr(1, string.length - 1);
  let silly = sillyBandz(spray, rainbow);
  return silly + spray.toUpperCase();
}

function sillyBandz(friend, bracelet){
  let memories = "abc".charCodeAt(1) - friend.charCodeAt(0);
  let ruined = sillyMe(memories);
  function sillyMe(why){
      let again = bracelet.indexOf(friend);
      return memories + again + why;
  }
  return bracelet.substr(0, ruined) + ruined;
}
```

What output is produced by a call to **`SillyString("agreatpaint")`**?

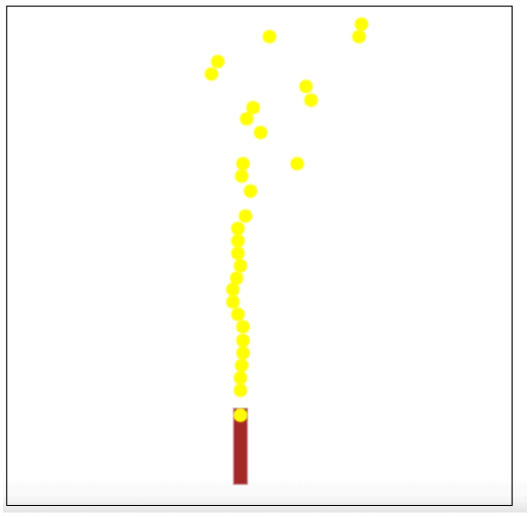
🎨 *Main Course Presentation: Graphics Melange*

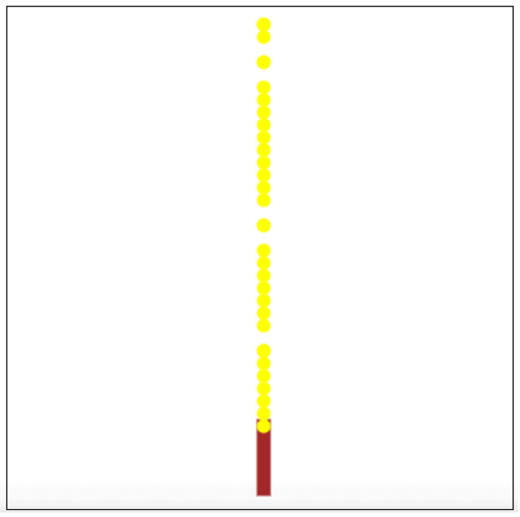
1) 🪄 **Magic Wand**

To simulate a magic wand on the mouse cursor, we'll draw a rectangle that follows the user's mouse and emits yellow sparks (circles) that float upwards. The top-left corner of the wand is attached to the mouse in both directions, and the wand is a brown rectangle with dimensions `WAND_WIDTH` by `WAND_HEIGHT`. Each `LAUNCH_TIME_STEP` milliseconds, a yellow circle with radius `SPARKLE_RADIUS` (which equals half the wand's width) should appear at the top center of the wand, and move upwards with velocity `SPARKLE_Y_VAL`, which is already negative. Note that at the time of its creation, the spark will share the same top-left coordinate as the wand.

Along with an event listener to track the mouse, note that you'll need nested animation interval timers: one to create each flying spark, and another to keep moving that particular spark, with an interval delay of `FLYING_TIME_STEP` milliseconds. Additionally, when the spark moves off the top of the screen, you should both remove the spark from the `GWindow` and stop that spark's animation with `clearInteval`.

You can start the wand with its upper-left coordinate at `(0,0)`. Screenshots of the program are displayed below. And for this assignment, unlike your Breakout! game, it's fine if the wand leaves the screen when the user's mouse does.





Fill out your implementation below, and feel free to use any of the provided constants.

```
/*
* File: magicWand.js
* -----------------
* This program implements a magic wand rectangle that
follows the user's mouse in both directions, and
continually shoots circular sparks toward the top.
*/

"use strict";

const GWINDOW_HEIGHT = 400;
const GWINDOW_WIDTH = 400;
const WAND_WIDTH = 10;
const WAND_HEIGHT = 60;
const SPARK_TIME_STEP = 2; // in milliseconds
const LAUNCH_TIME_STEP = 10;  // in milliseconds
const SPARK_Y_VELOCITY = -10;
const SPARK_RADIUS = 5;

function magicWand(){
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);

   // fill this in along with any helper and callback
   functions.

















}
```

🧵 *String Theory*

1) Write a function **`removeDoubledLetters`** that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter, e.g.,

- ☐ **`removeDoubledLetters("bookkeeper") => "bokeper"`**
- ☐ **`removeDoubledLetters("zzzz") => "z"`**
- ☐ **`removeDoubledLetters("verrycoool") => "verycol"`**

```
function removeDoubledLetters(str){
    // write your implementation here

}
```

2) Write a function **`longestVowelStreak(str)`** that takes in a string **`str`** and returns the longest number of vowels that appear in a row.

We'll consider vowels to be only the letters **`A, E, I, O, U`**. If str does not contain any vowels (or any characters at all for that matter), you should return 0. Furthermore, your count should be case-insensitive—meaning your function does not distinguish between uppercase or lowercase letters when counting vowels.

Here are a few sample calls of the function, with the longest vowel streak underlined.

- ☐ **`longestVowelStreak("syzygy") => 0`**
- ☐ **`longestVowelStreak("QUEUE") => 4`**
- ☐ **`longestVowelStreak("Jouaeint means play in French") >= 5`**

Hint: To check if a character is **`A, E, I, O, U`**, you can check if the character resides in the string **`"AEIOU"`**.

```
function longestVowelStreak(str){
    // write your implementation here

}
```

🌤️ *Aren't you Array of Sunshine*

1) Write a function **`interleaveArrays`** that takes in two arrays of integers **`arr1`** and **`arr2,`** and returns a new array that interleaves the elements from **`arr1`** and **`arr2`**. That is, the result should be an array that contains—in order—alternating elements / integers from **`arr1`** and **`arr2`**, starting with a number from **`arr1.`**

```
let arr1 = [1, 2, 3, 4];
let arr2 = [10, 20, 30, 40];
let newArr = interleaveArrays(arr1, arr2);
// newArr should be [1, 10, 2, 20, 3, 30, 4, 40]
```

If one of the arrays passed in is larger than the other, any remaining elements from the larger list which have not been included yet in the interleaved list are simply added (in order) to the end of the result. Additionally, if one of the lists passed in is empty (**`[]`**), the result is simply the other list. If both lists are empty, the result is the empty list.

```
let arr1 = [100, 200];
let arr2 = [1, 2, 3, 4, 5, 6];
let newArr = interleaveArrays(arr1, arr2);
// newArr should be [100, 1, 200, 2, 3, 4, 5, 6]
```

```
let arr1 = [1, 2, 3, 4, 5, 6];
let arr2 = [100];
let newArr = interleaveArrays(arr1, arr2);
// newArr should be [1, 100, 2, 3, 4, 5, 6]
```

Write your **`interleaveArrays`** implementation below (of course, you'd have much more space were this the real midterm paper).

```
function interleaveArrays(arr1, arr2){
     // write your implementation here
}
```

2) With the US National Debt now well over 37 trillion dollars 💵, you decide to create the notion of "Big Numbers" in JavaScript, allowing you to represent an arbitrarily large positive integer value **using an array of digits**. For instance, we could represent the national debt 37,918,570,029,078 (at a certain timestamp I saw last night) as

```
[3,7,9,1,8,5,7,0,0,2,9,0,7,8]
```

However, to allow you to more easily add such numbers, you **store the digits of the "Big Number" in reverse**, such that the ones digit is in index 0, the tens digit is in index 1, and so forth. So the debt number above would actually be represented as

array	8	7	0	9	2	0	0	7	5	8	1	9	7	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Note that the array should have exactly as many elements as the number of digits in the number it represents. Suppose now we have a `bigNum1` representing 9564 and `bigNum2` representing 867, and we want to compute the sum of both numbers.

bigNum1	4	6	5	9
index	0	1	2	3

bigNum2	7	6	8
index	0	1	2

We can add the ones digits in the 0th index, then the tens digits in the 1st index, and so forth, to produce the resulting sum 10431, as shown in the variable result below.

result	1	3	4	0	1
index	0	1	2	3	4

Your job is to implement a function **`addBigNumbers`** that accepts two "Big Numbers" (arrays of digits in backward order), and returns a new "Big Number" array that represents the sum of the two numbers passed in, using the algorithm described above to add the numbers digit by digit, propagating any necessary carries to the next digit. Your function should not modify the arrays passed in.

```
function addBigNumbers(bigNum1, bigNum2){
    // write your implementation here
}
```

🍰 🎂 *Would you care for some dessert? No object(ion) your honor*

1) 🥣 **Great CS106AX Bake Off**

This is fiction btw. We as a class are holding a great 106AX bake off contest, in the spirit of the popular British TV show. To prepare, you collect several ingredients, and place them in a pantry, as represented by the JavaScript object below. The names or keys are ingredients, and the values are amounts—metric system, of course.

```
let pantry = {
     flour: 400,
     sugar: 200,
     salt: 10,
     chocolate: 100,
     vanilla: 50,
};
```

To prepare, you will make several cake recipes / orders, which are placed in an array of objects as displayed below. Each object inside, similar to the pantry, has keys that are ingredient names and the values are the amounts needed to bake the cake.

```
let cakeOrders = [
    {flour: 40, sugar: 20, vanilla: 4]},
    {flour: 20, salt: 10, chocolate: 5, vanilla: 5},
    {flour: 60, sugar: 40, chocolate: 10},
    ... more cake orders
];
```

Our task is to write two functions to help with the baking marathon. The first function, **`canMakeOrders`**, should take in a pantry object and list of cake orders as input, as structured above, and return a boolean variable that is **`true`** if the pantry has sufficient ingredient amounts to **make all the orders**, and **`false`** otherwise. You can assume that if an ingredient name is not in the pantry object, its amount is zero. Furthermore, **`canMakeOrders`** should not directly modify the pantry object (you can make a copy of the object inside if needed).

```
function canMakeOrders(pantry, cakeOrders){
     // write your implementation here
}
```

Lastly, we'll write a function **`fulfillOrders`** , which should modify the pantry object in place, and decrement as many ingredient amounts as the cake orders together require. You can assume here that the pantry can fulfill all the cake orders. Your function should also return the modified pantry object.

```
function fulfillOrders(pantry, cakeOrders){
    // write your implementation here

}
```

A short example run of both cake-baking functions is shown below.

```
>> let pantry = {flour: 100, sugar: 50, salt: 20};

>> let cakeOrders = [
   {flour: 40, sugar: 20, salt: 10},
   {flour: 30, sugar: 20, salt: 5},
];

>> canMakeOrders(pantry, cakeOrders);
true

>> pantry = fulfillOrders(pantry, cakeOrders);

>> console.log(pantry); // updated pantry stockpile
{flour: 30, sugar: 10, salt: 5}

>> let newCakeOrders = [
    {flour: 30, sugar: 15, salt: 5, chocolate: 5},
];

>> cakeMakeOrders(pantry, newCakeOrders)
false
```