CS 106AX Programming Methodologies in JavaScript and Python Autumn 2026

IG Problem 3A: Fanfiction Ship Names

Written by Ben Yan

In fandom, a *ship* is the pairing of two individuals in a story—or occasionally, real life—that fans would love to see in a romantic relationship. Maybe they're already in one, who knows ☺. For instance, Korra and Sato in *Avatar: The Legend of Korra*, Dean and Castiel in *Supernatural,* and Kylo and Rey in the *Star Wars* films.

Many ships are celebrated with a *ship name*, a combined title that's typically formed by some linguistic marriage of their names or clever wordplay. For this problem, we'll implement a quirky algorithm:

```
function shipThem(nameOne, nameTwo)
```

which creates a ship name between two first names, **`nameOne`** and **`nameTwo`**, as follows.

- If **`nameOne`** begins with a consonant, we take everything before the first vowel in **`nameOne`**, i.e. the starting stream of consonants, and concatenate it with everything including and after the first vowel in **`nameTwo`**. For instance, **`shipThem("draco", "harry")`** would result in **`"drarry".`** In the event that **`nameTwo`** starts with a vowel, e.g., **`"amity"`**, we include all of it in the ship name, e.g., **`shipThem("luz", "amity")`** becomes the delightful **`"lamity"`**.

- If **`nameOne`** begins with a vowel, we take everything before the **second** vowel in **`nameOne`**, and concatenate it with everything after (**but not including**) the first vowel in **`nameTwo`**. For instance, **`shipThem("elio", "oliver")`**, the leads from *Call Me by Your Name*, becomes **`"elliver".`**

You may assume every input is lowercase, a single word, and has at least one consonant and vowel. If the name starts with a vowel, you can assume there will be a second one. We consider *a, e, i, o, u* to be vowels, but not *y*—so sorry linguists! Here are some additional fan and real ships if you're into that ☺.

JavaScript Console

```
> shipThem("bruce", "selina")
brelina
> shipThem("megumi", "yuji")
muji
> shipThem("wednesday", "enid")
wenid
> shipThem("armin", "eren")
armren
>
```



Use the space on the next page to present your implementation of **`shipThem`**, and any helper functions. My approach makes use of at least two, one of which is provided to you, **`isVowel`**, and the other which you're encouraged to write yourself, **`findFirstVowel(word, start)`**. This function can return the index of the first vowel in **`word`** at or after the **`start`** index. To search the entire word, omit the **`start`** parameter.

```
/* This function takes two one-word names, and smashes them together into
 * a fanfic-ready ship name, according to the vowel / consonant rules above.
 */
function shipThem(nameOne, nameTwo) {









}

/* Returns index of first vowel at or after the start position, and -1 if
 * no vowel could be found. If the start is omitted, it's assumed to be 0.
 */
function findFirstVowel(word, start) {
      if (start === undefined) start = 0; // complete this implementation!









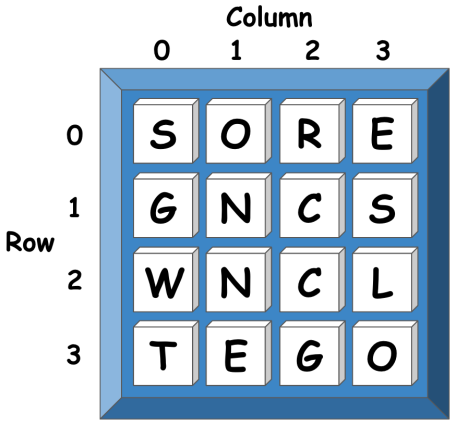
}

/* returns true if the string is of length one, and a lowercase vowel */
function isEnglishVowel(ch) {
      return ch.length === 1 && "aeiou".indexOf(ch) >= 0;
}
```

IG Problem 3B: Boggle – The Mind-Boggling Word Game

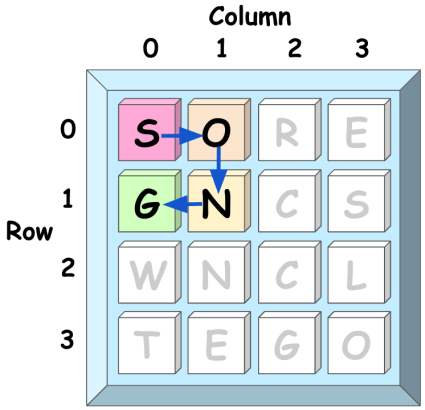
Written by Ben Yan
(a homage to a legendary assignment in CS106X, the Ozymandias of Jerry's great, storied CS classes)

If you're a millennial like me, perhaps you're familiar with the word game *Boggle,* which is great for traveling and killing time on long, long road trips, when one can help but mutter, *are we there yet Mom?*



Unlike the vocabulary games that kids these days play in the cold, distant blue light of their phones, cough *Wordle*, *Boggle* is waged over a real, physical game board—upon which a grid of letter dice is distributed. The grid is rectangular, typically 4x4 or 5x5. **Your goal is to find all the words you possibly can by tracing a path through adjoining letters**, **i.e. letters that are adjacent horizontally, vertically, or even diagonally.** Longer words are worth more points, of course ☺.

Suppose you're playing *Boggle* with your frenemy, Ben, and they boast they've found a new word. You're skeptical, not because you think they haven't touched a dictionary or read a book in their life—but as you believe they're messing with you, so you decide to investigate ☺. Write the function **`validateWord`**, which accepts three arguments as input:



***SONG** on the board.*

- **`grid`**: An array of strings, where each string corresponds to the letters in a row. The letter at a given **`row`** and **`column`** is given by **`grid[row][column]`**. As an example, for the Boggle board visualized on the right, the **`grid`** parameter would appear as:

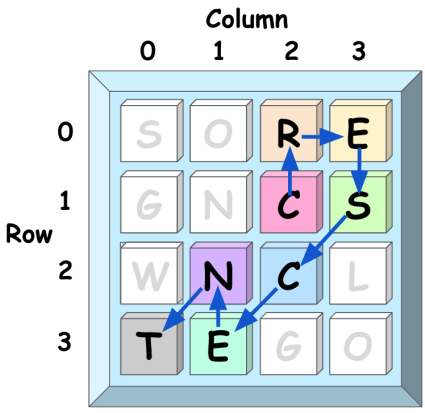
  `["SORE", "GNCS", "WNCL", "TEGO"]`

- **`word`**: A string whose existence we want to verify on the provided grid.

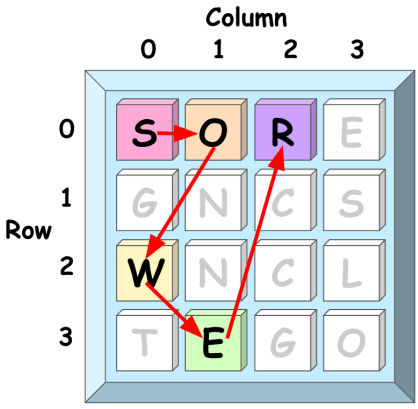
- **`path`**: An array of two-element arrays, where each element has form **`[row, column]`**. This is an ordered sequence of letter squares on the grid that potentially traces out a word. For instance, the word **`"SONG"`** in the grid above possesses the path **`[[0,0], [0,1], [1,1], [1,0]]`**.

We return **true** if the **`path`** traces the input **`word`** on our Boggle **`grid`**, without violating any game rules, and **false** otherwise. Specifically, the following conditions must be true for validity.

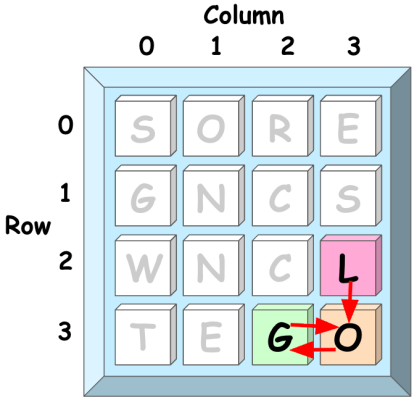
- The length of the **`word`** string must equal the length of **`path`**.
- The **`word`** string must perfectly match the letter squares traced by the **`path`** on the **`grid`**.
- Each move between consecutive squares in the **`path`** must be valid, i.e. the squares must be adjoining horizontally, vertically, or diagonally. For instance, from the **`"N"`** at **`[1,1]`**, you can move to 8 possible squares next: **`[0,0]`**, **`[0,1]`** , **`[0,2]`** , **`[1,0]`** , **`[1,2]`** , **`[2,0]`** , **`[2,1]`**, or **`[2,2]`**.
- Lastly, the **`path`** is composed entirely of distinct squares, i.e. no square is visited twice. The same letter, however, can be repeated if the instances appear on different squares.



***CRESCENT** on the board, the longest word I could find.*



*Failed attempt at **SOWER,** the title of a Kemble Scott novel.*



*Failed attempt at **LOGO**, where Steph Curry shoots from.*

Given the example `grid = ["SORE", "GNCS", "WNCL", "TEGO"]`, here are some successful inputs.

- `validateWord(grid, "SONG", [[0,0], [0,1], [1,1], [1,0]]) => true` ✓
- `validateWord(grid, "CRESCENT", [[1,2], [0,2], [0,3], [1,3], [2,2], [3,1], [2,1], [3, 0]]) => true` ✓

Here is a pair of inputs that returned `false` instead, due to violating a game condition.

- `validateWord(grid, "SOWER", [[0,0],[0,1],[2,0],[3,1],[0,2]]) => false`. The path features two illegal moves—the skydive from `"O"` to `"W"`, and the `"E"` to `"R"` levitation.
- `validateWord(grid, "LOGO", [[2,3], [3,3], [3,2], [3,3]]) => false`. While each move individually is valid, the path self-intersects, visiting the `"O"` square at `[3,3]` twice.

Here a few notes to help guide your implementation:

- You have no guarantee about the grid size, other than it has at least 2 rows and 2 columns.
- You may assume the `word` input is a valid English word. You don't need an actual dictionary here.
- You may assume `path` has length at least 3, and that each `[row, column]` element is a location within grid bounds. That is, you do not need to perform out-of-bounds error-checking for the `path`.
- *Hint:* To check if a move between two different squares is valid, the row numbers can be at most one apart, and the column numbers can be at most one apart. You're welcome to use `Math.abs()`.
- **To check that no squares are repeated in `path`**, while you could use a data structure to keep track of previously visited squares, **I recommend that you loop through each pair of elements, checking if any have the same row and column number** (note that `[6,7] !== [6,7]`, as `===` checks whether the two arrays refer to the same exact data in memory, not that their contents are identical).

Use the next page to present your shimmering implementation, and any helper functions your heart desires.

```
/* Returns true if the path of squares traces the given word on the Boggle
 * grid, without violating any of the game conditions, and false otherwise.
 */
function validateWord(grid, word, path) {









}
```