All Anagrams

Learn how to determine if multiple strings are all anagrams of each other. While it seems as if this would require a fairly poor time and space complexity, we can come up with clever solutions that make this problem tractable.

All Anagrams

Instructions

Write a function that takes in an array of strings. Return true if all strings are anagrams of one another and false if even a single string is not an anagram of the others.

Input: Array of Strings

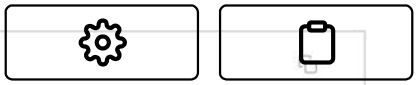
Output: Boolean

Examples

```
allAnagrams(['abcd', 'bdac', 'cabd']); // true
allAnagrams(['abcd', 'bdXc', 'cabd']); // false
```

Hints

- Think about what it means for two strings to be anagrams. They should all have the same characters present in the same number, perhaps in a different order.
- It would make sense to express the time complexity in terms of two variables.



```
function allAnagrams(strings) {
    // Your code here
}
```

Solution 1

```
function allAnagrams(strings) {
    const sortedStrings = strings.map(str => str.split('').sort().join(''));

    for(let i = 1; i < strings.length; i++) {
        if(sortedStrings[i] !== sortedStrings[0]) {
            return false;
        }
    }

    return true;
}
```

Show Results Show Console

1.05s

4 of 4 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	['abcd', 'bdac', 'cabd']	true	true	Succeeded
✓	['abcd', 'bdXc', 'cabd']	false	false	Succeeded

Result	Input	Expected Output	Actual Output	Reason
✔	['123', '132', '213', '231', '312', '321 …	true	true	Succeeded
✔	['123', '122']	false	false	Succeeded

How it Works

Line 2 above performs several functions:

1. Turns each string into an array of characters
2. Sorts each array of characters
3. Joins the character array back into a sorted string

If all original strings are anagrams, they should all be exactly the same string in our new array.

The for-loop checks to make sure all strings are identical. If not, immediately return `false`. If we get to the end, we can return `true`.

Time Complexity

Line 2

We'll have to describe the time complexity in terms of two factors: the length of the array, `a`, and the length of each string, `s`.

We'll first worry about a portion of the code on line 2:

```
str.split('').sort().join('')
```

`s` is the length of `str`. We have three operations to consider, a split, a sort, and a join.

Split and join are both `O(s)` operations - they have to process every character in a linear fashion.

Sorting the character array is an `O(s * log(s))` operation.

For this statement, we have a total complexity of:

```
O(s + s * log(s) + s)
= O(2s + s * log(s))
=> O(s * log(s))
```

We can now consider all of line 2. The code we've considered is `O(s * log(s))`. This code is inside the `map` statement, which is a loop going through the array. The complexity of the loop is `O(a)`, where `a` is the length of the array.

Multiplying these two gives us a final value of

```
O(a * s * log(s))
```

for line 2.

for-loop

The for-loop is a standard `O(n)`, or in our case `O(a)`, operation. The comparison inside on line 5 is linear in terms of the length of the strings, `s`. The total time complexity for the loop is `O(a * s)`.

Adding this to the value for line 2 gives us `O(a * s + a * s * log(s))`. Since the 2nd term is higher magnitude, we can drop the first term, which yields:

O(a * s * log(s)).

Space Complexity

We're storing values in the `sortedStrings` array. We're storing essentially every string we get. This gives us space complexity of:

O(a * s).

Solution 2

```
function getCharCount(str) {
    const charCount = {};

    for(let i = 0; i < str.length; i++) {
        const char = str[i];

        if(charCount[char] === undefined) {
            charCount[char] = 1;
        }else {
            charCount[char]++;
        }
    }

    return charCount;
}

function allAnagrams(strings) {
    if(strings.length === 0) {
        return true;
    }

    for(let i = 1; i < strings.length; i++) {
        if(strings[i].length !== strings[0].length) {
            return false;
        }
    }

    const firstCharCount = getCharCount(strings[0]);

    for(let i = 1; i < strings.length; i++) {
        const thisCharCount = getCharCount(strings[i]);
```

Show Results Show Console

1.13s

4 of 4 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	['abcd', 'bdac', 'cabd']	true	true	Succeeded
✓	['abcd', 'bdXc', 'cabd']	false	false	Succeeded
✓	['123', '132', '213', '231', '312', '321 …	true	true	Succeeded
✓	['123', '122']	false	false	Succeeded

How it Works

Helper Function

At the top, we have a helper function that takes in a string and returns an object with the counts of all characters in the string. For example, a string of `hello` would give us the object

```
{
    h: 1,
    e: 1,
    l: 2,
    o: 1
}
```

Main Function

We start the function `allAnagarams` by returning `true` for an empty array. We then check to make sure all strings are the same length, on line 22. If not, we can return `false`.

On line 28, we invoke our helper function and pass in the first string in our array. We'll use the object we get back, `firstCharCount`, to compare the rest of the strings in the array against.

In the final for-loop (line 30), we go through every string in our array. At each iteration, we generate a character count object. The for-in loop makes sure that this object has the same key-value pairs as our template object, `firstCharCount`. If any values are off, we know that the strings aren't anagrams and we can return `false`.

Once we get through all strings, we can return `true`.

Time

We'll again define the length of the array as `a` and the length of the strings as `s`.

The first for-loop in `allAnagrams` is `O(a)`, as we go through the array but don't worry about the characters.

Line 28 calls the function `getCharCount`, which processes every character in the string it receives. `getCharCount` is therefore `O(s)`.

The for-loop goes through every string in our array except the 1st, so we can consider it `O(a)`.

Inside the loop, the call to `getCharCount` in the loop is `O(s)`. The for-in loop on line 33 again goes through every character in the string currently being processed. It's, therefore, `O(s)`.

Since the second `O(s)` call is after the first, we can add them up, bringing the time complexity of lines 31 - 37 to `O(2s)`. Multiplying this with the for-loop yields `O(2s * a)`, which simplifies to:

O(a * s)

Space

We only ever store two objects in memory at once throughout this function - `firstCharCount` and `thisCharCount`. The space complexity is therefore `O(2s)`, simplifying to:

O(s)

← Back

Single Mutation

Next →

Rotate Matrix

Mark as Completed

Report an Issue