Is Unique

We'll discuss three ways to determine if all characters in a string are unique. Each has benefits and drawbacks regarding time and space complexity.

Is Unique

Instructions

Create a function that determines whether all characters in a string are unique or not. Make it case-sensitive, meaning a string with both `'a'` and `'A'` could pass the test.

Input: String

Output: Boolean

Examples

```
isUnique('abcdef'); // -> true
isUnique('89%df#$^a&x'); // -> true
isUnique('abcAdef'); // -> true
isUnique('abcaef'); // -> false
```

Immediately, we know that we'll have to process every character in the string. The best time complexity possible for this algorithm is linear, or `O(n)`. There's no way to get around that.

```
function isUnique(str) {
    for(let i = 0; i < str.length; i++) {
        if(str.lastIndexOf(str[i]) != i)
```

```
            return false;
        }
    return true;

}
```

Show Results   Show Console

1.13s

5 of 5 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	'abcdef'	true	true	Succeeded
✓	'89%df#$^a&x'	true	true	Succeeded
✓	'abcAdef'	true	true	Succeeded
✓	'abcaef'	false	false	Succeeded
✓	'1233abc'	false	false	Succeeded

Solution 1

Here's the brute-force, simplest solution.

```
function isUnique(str) {
    for(let i = 0; i < str.length; i++) {
        if(str.lastIndexOf(str[i]) !== i) {
            return false;
        }
    }

    return true;
}
```

How it Works

We go through the entire string. At each character, we make sure that the final index of the character is the same as the index our loop is currently on.

If this check fails, we know that the letter appears again further down the string. We can immediately return `false`. If we get to the end, we've ensured that all characters are unique and can return `true`.

Time

We have a for-loop that's going through every letter in the string, so we start with `O(n)`.

Inside the loop, the call to `str.lastIndexOf()` runs through our string backward. It's another loop. When we're on our first letter, assuming the string has only unique characters, it'll run through the entire string from back to front.

As we approach the end of the string, `lastIndexOf()` has to traverse fewer characters to get to our index. On average, `lastIndexOf` will go through half the length of the string on each iteration. This means `lastIndexOf` has a time complexity of `O(1/2 * n)`, which becomes `O(n)` after dropping the constant.

Since this call to `lastIndexOf()` is nested inside our for-loop, we multiply their time complexities, bringing our final time complexity to

O(n^2).

Space

No matter how large the string, we only ever use one variable in our function: `i`. This shows us that our function uses constant space, or:

O(1)

This is the best space complexity achievable.

For time complexity, we can do better.

Solution 2

```
function isUnique(str) {
    const chars = str.split('').sort();

    for(let i = 1; i < chars.length; i++) {
        if(chars[i] === chars[i - 1]) {
            return false;
        }
    }

    return true;
}

```

How It Works

We insert every character present in the string into an array and then sort the array. Since the array is sorted, identical characters will appear next to each other.

We then go through the sorted array one by one and check if the character is the same as the one before it. If so, we return `false`. If we process the whole array, we can return `true`.

Time

The complexity of a sorting algorithm can be approximated as `O(n * log(n))`. The loop scales linearly, giving us `O(n)`.

Since these two processes happen apart from one another, we can add their time complexities to get a value for the whole function. This gives us `O(n + n * log(n))`. Dropping the lower order term, we get:

O(n * log(n)).

This is the final time complexity.

Space Complexity

We need to store every character in an array, so our space complexity is:

O(n).

We can actually do even better.

Solution 3

```
function isUnique(str) {
    const chars = {};

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

        if(chars[thisChar]) {
            return false;
        }

        chars[thisChar] = true;
    }

    return true;
}
```

How it Works

As we process each character in our loop, we insert it into our object. This lets us keep track of every character we've already encountered.

At each character, we check if our object contains the character and if it does, we return `false`. Otherwise, we add the character to the object and continue to the next index. At the end, we return `true`.

Once we get through the whole string, we know that each character is unique and we can return `true`.

Time

We have a for-loop that's going through every letter in the string, so we start with `O(n)`.

Inside the loop, we perform actions that are all `O(1)`. We have no loops and no loop-like functions. Object insertions and retrievals are `O(1)`, as they generally act like a hash table (https://codeburst.io/objects-and-hash-tables-in-javascript-a472ad1940d9) in JavaScript. So, our final time complexity is:

O(n).

Space

We're inserting every character we come across into a set. Therefore, our space complexity is:

O(n).

The amount of space we use is directly proportional to the size of our input.

Improvements

There's a way to make this last solution slightly more elegant. Instead of an object, we can use an ES2015 construct, the *Set (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)*.

A set is similar to an object. We insert items into the set and retrieve them later.

One of a set's main strengths is that it can only store one copy of each item. Attempts to add an item twice fail silently, so it only stores unique values.

A set also has instant insertion and retrieval of items.

```
function isUnique(str) {
    const chars = new Set();

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

        if(chars.has(thisChar)) {
            return false;
        }

        chars.add(thisChar);
    }

    return true;
}
```

I consider this slightly more elegant because it uses a data structure, the Set, in the exact manner it was intended for. The method names (`has` and `add`) are descriptive and, overall, the function becomes a bit easier to understand.

Believe it or not, the Set actually allows us to transform this code into a one-line function. This is the “clever” solution. However, it’s not quite as efficient as Solution 3.

Solution 4

```
function isUnique(str) {
    return new Set(str).size === str.length;
}
```

If all characters are unique, the size of the set will be the same length as the string. If we have duplicate characters, the set will be smaller. This is because the set completely ignores duplicate insertions.

While time and space complexity are both also `O(n)` for this solution, it is, in general, slightly slower than Solution 3. This is because Solution 3 has a mechanism to short-circuit the loop if we come across a repeat character. This solution, on the other hand, must go through the entire string and then return an answer.

Arrays

Believe it or not, Solution 4 and both versions of Solution 3 above work for strings *and arrays*. Don't believe me? Click the `Test` button below.

```
function isUnique(str) {
    return new Set(str).size === str.length;
}
```

Strings and arrays are both iterable items that the set treats in the same manner.

Conclusion

Brute Force Solutions

The first solution is the one most people come up with. It's simple and straightforward. If you can identify the time complexity and then work to make it better, you'll ace this question in an interview.

The second solution is better and improves our time complexity at the cost of space.

Ideal Solutions

The third solution is an example of when a data structure greatly reduces our time complexity. By introducing an object or set, we turned an `O(n * log(n))` solution to an `O(n)` solution, a significant improvement.

While the final solution is clever and very simple, it isn't as efficient as the third. Still, during an interview, it would be an excellent answer.

Extending our Code

The fact that our last solution happens to work for arrays as well as strings shows us that our solutions can easily be modified to work for other types of input. For many problems in this course, only minor adjustments would be necessary.

Learning the techniques used to solve these problems is the main goal. Once you have the techniques, you'll be able to apply them to a range of problems.

Takeaways

Take away the idea that applying various strategies to a problem can significantly improve time or space complexity. Sorting our data brought our time complexity down significantly.

Using an appropriate data structure can also improve time complexity. Try using an object, array, set, stack, or queue to store your data and see if that opens up any new avenues for your solution.

← Back
Welcome

Next →
Flatten Array

✓ Mark as Completed

Report an Issue