Highest Frequency

In this problem, we'll learn how to keep a count of values as we go through an array. From that data, we'll return the most common string in the array.

Highest Frequency

Instructions

Write a function that takes an array of strings and returns the most commonly occurring string in that array.

If there are multiple strings with the same high frequency, return the one that finishes its occurrences first, while going left to right through the array.

Input: Array of Strings

Output: String

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

Solution

```
function highestFrequency(strings) {
    const frequencies = {};
    let maxFrequency = 0;
    let mostFrequentString = strings[0];

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

        if(frequencies[thisStr] === undefined) {
            frequencies[thisStr] = 1;
        } else {
            frequencies[thisStr]++;
        }

        if(frequencies[thisStr] > maxFrequency) {
            maxFrequency = frequencies[thisStr];
            mostFrequentString = thisStr;
        }
    }

    return mostFrequentString;
}
```

How it Works

Variables

The three variables declared at the top of the function are meant to keep track of data throughout the loop below. `frequencies` will contain data for every string that we process at every iteration. The keys will be the strings we process and those keys' values will be their frequency.

Counting Frequency

`maxFrequency` and `mostFrequentString` will be maintained across iterations, updating when needed.

In the loop, we first check if we've already come across the string we're currently working with. If so, it already exists in our `frequencies` object and we can increment its value.

If we haven't come across it yet, we'll insert it into our object and give it a value of `1`.

Keeping it Current

After that, on line 15, we're checking to see if the value we just updated or inserted into `frequencies` is greater than the largest frequency we've seen so far. If so, we update both the highest frequency and most frequent string.

After processing the whole array, `mostFrequentString` will contain the correct value and we can return it.

(/learn)

Time

The time complexity of this function is:

O(n).

We're processing every item once. No operations in the loop depend on the size of the array, so they're all `O(1)`.

Space

We're keeping track of every string that we process, so space complexity in the worst case is also:

O(n).

Conclusion

This problem essentially forces us to store and update data as we go through our loop. This is a good technique to keep in mind.

Often, problems will not require this and a solution can be reached without keeping track as we do here. Applying this strategy, however, can sometimes reduce the time complexity of the algorithm.

In general, try seeing if an object or other data structure can fit in an algorithm. Storing data in a creative manner can increase algorithmic speed in unexpected ways.

← Back
Remove Dupes

Next →
String Rotation

✓ Mark as Completed

Report an Issue