Demystifying Big O

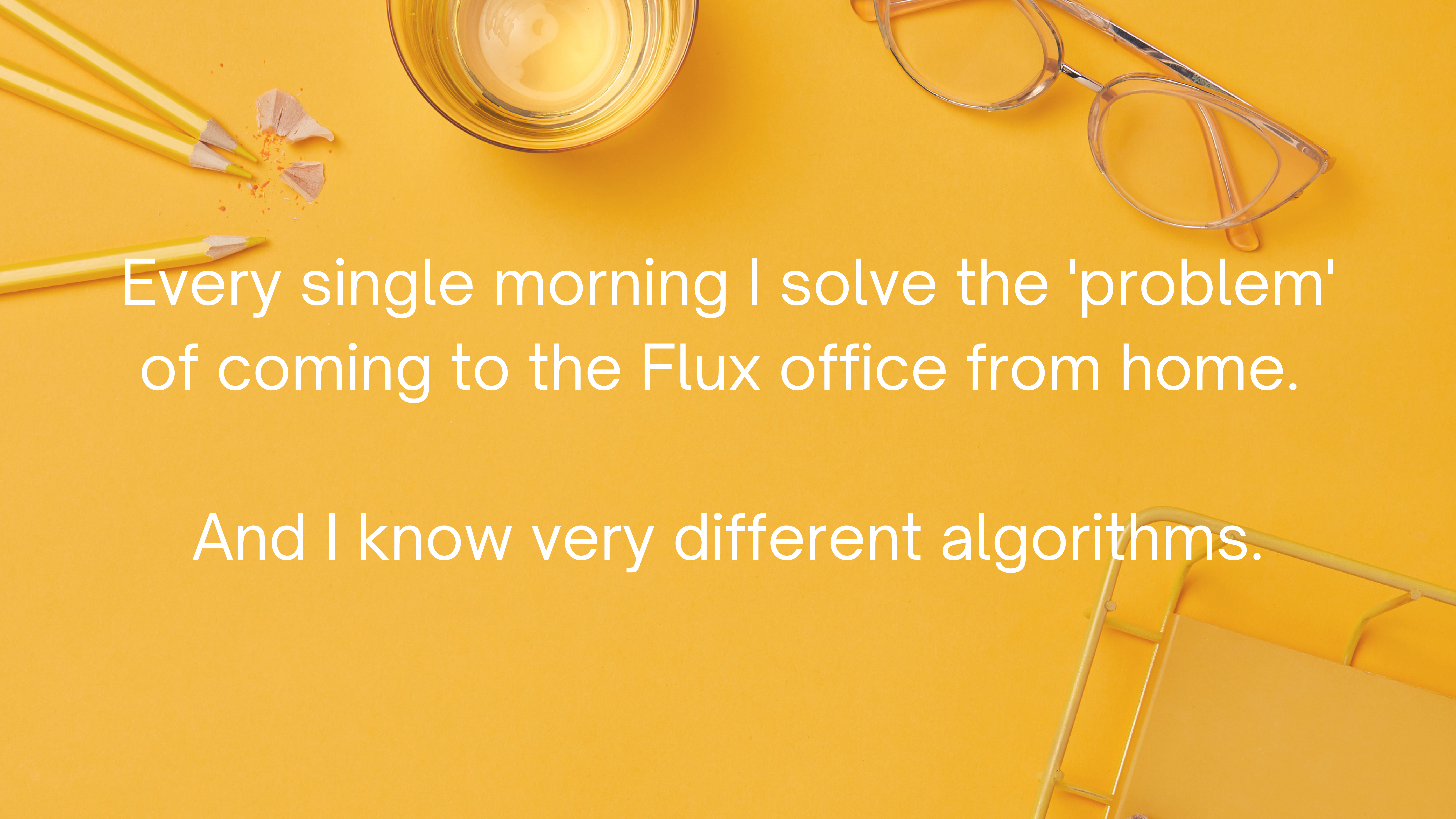
Computational Complexity

Is a theoretical branch of Computer Science

Is used to determine the cost of algorithms in:

- Time (count of instructions)
- Space

Algorithms are sorted into classes (like P and NP)



Every single morning I solve the 'problem' of coming to the Flux office from home.

And I know very different algorithms.

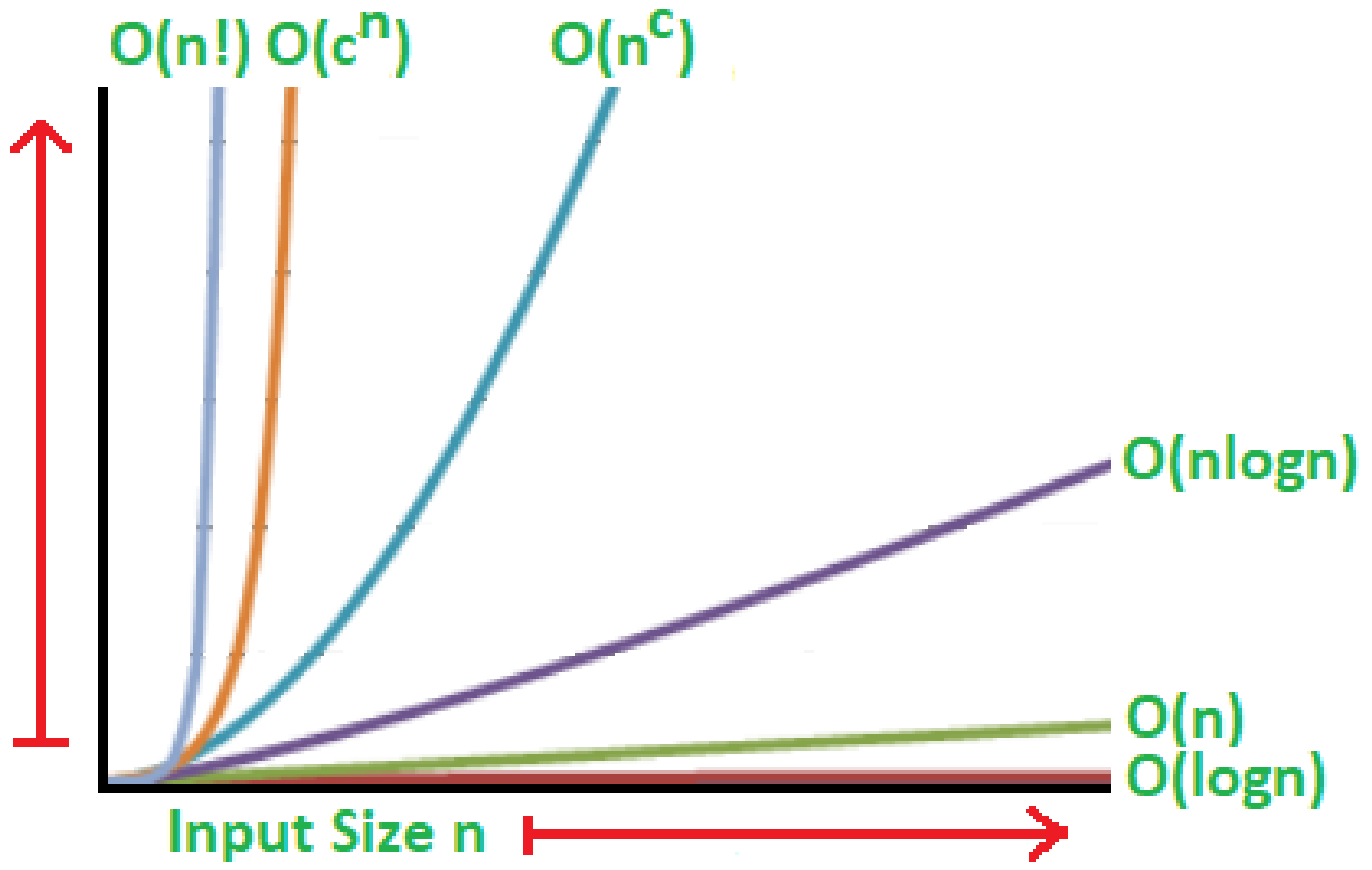
"In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows."

— Wikipedia

"Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation."

— Wikipedia

Common functions for Big O



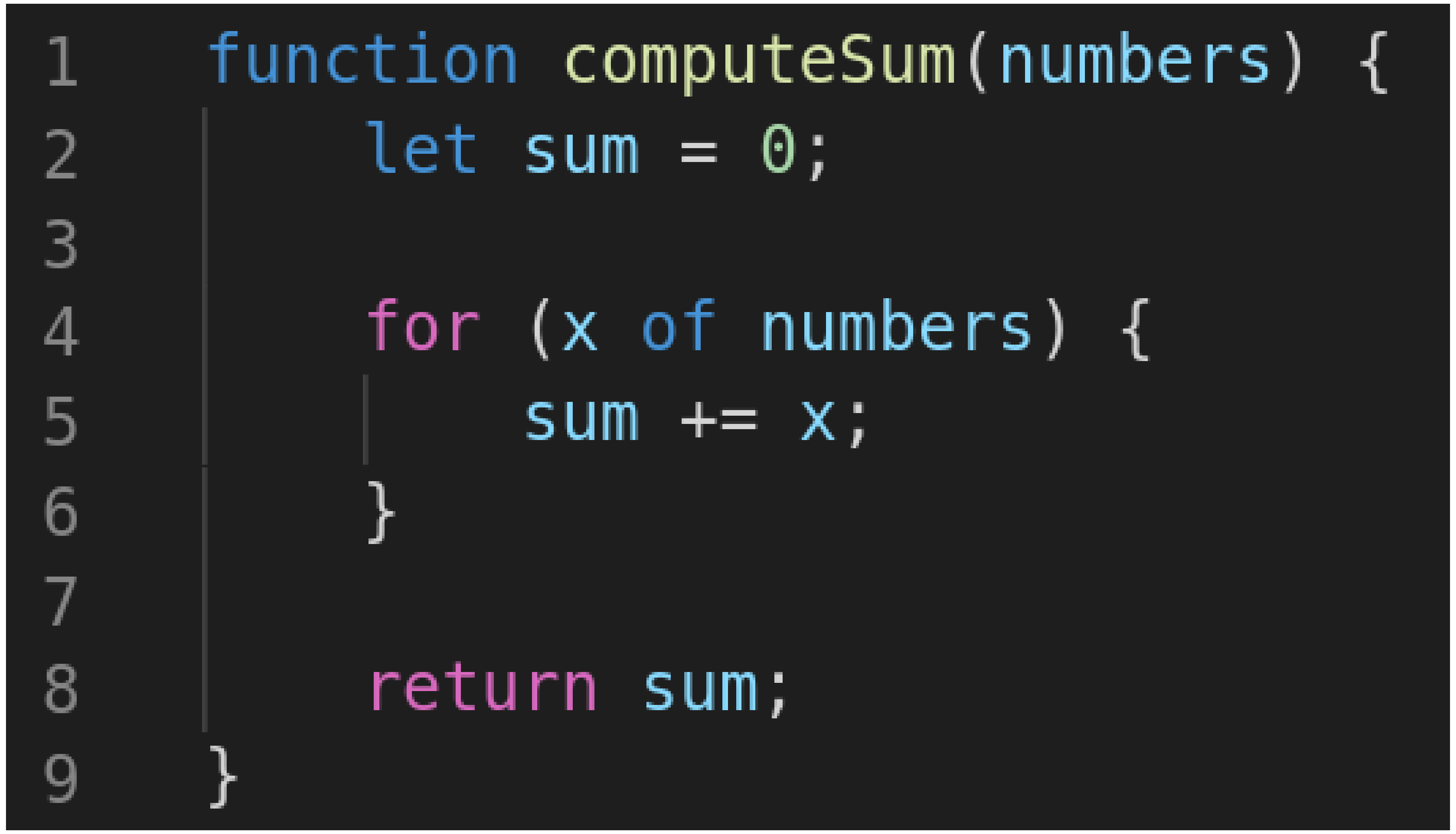
$O(n!), O(c^n), O(n^c)$ - Worst

$O(n\log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

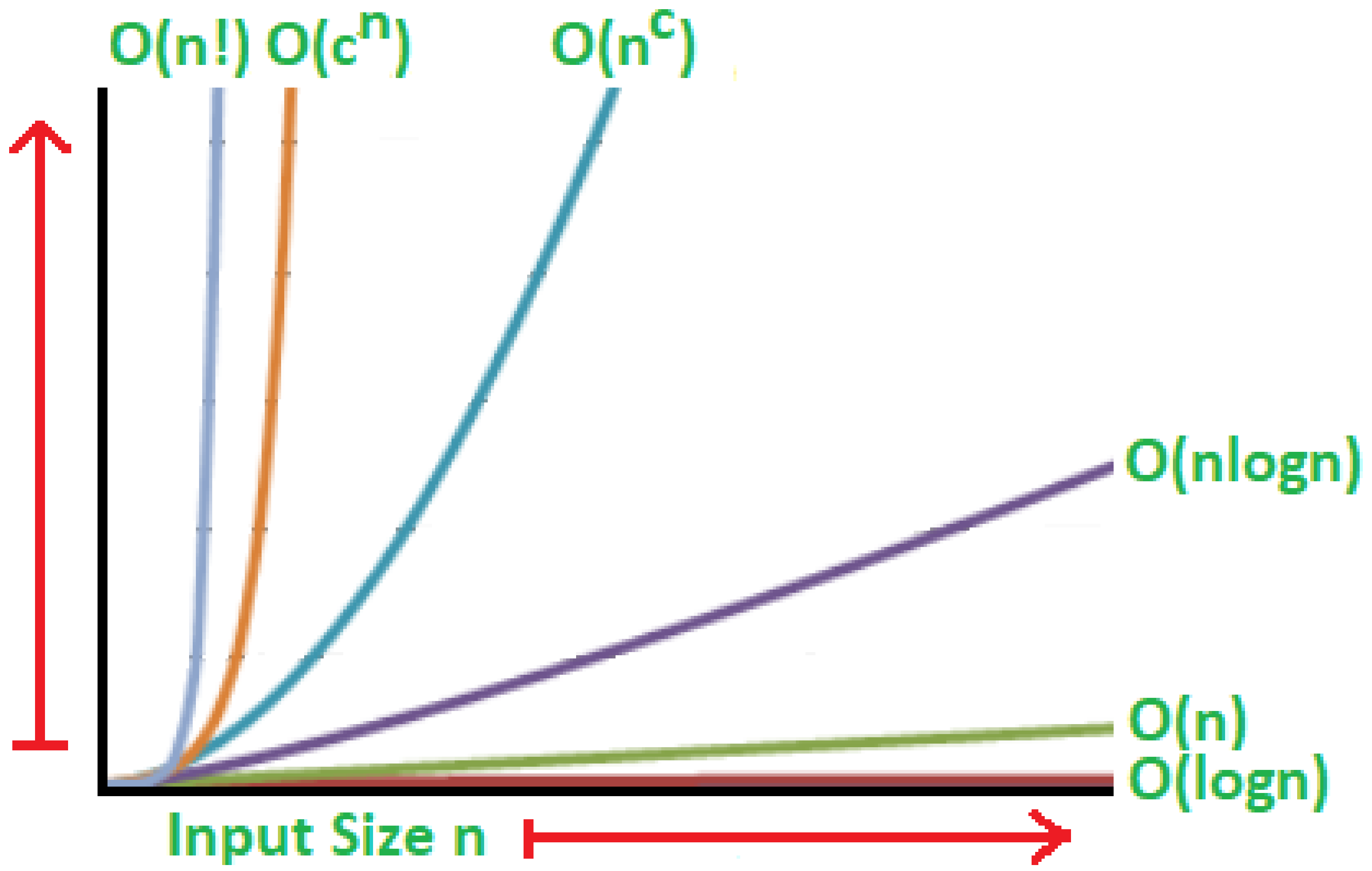


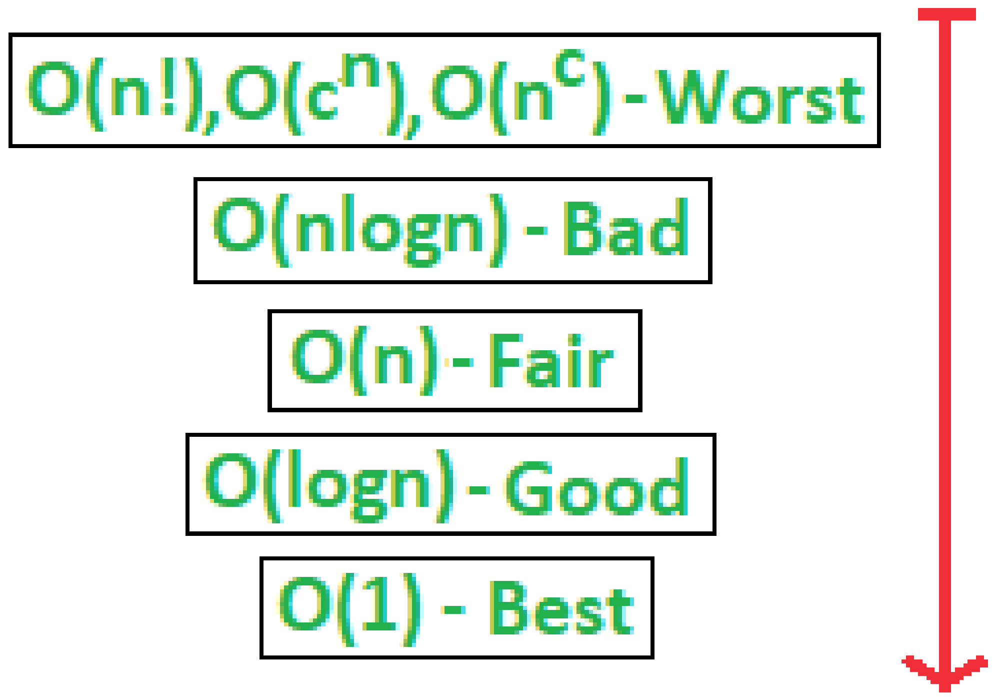
```
function computeSum(numbers) {
    let sum = 0;

    for (x of numbers) {
        sum += x;
    }

    return sum;
}
```

Space: O(1)

Time: n + 1 ~ O(n)

Common functions for Big O





$O(n!), O(c^n), O(n^c)$ - Worst

$O(n\log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

Basic Rules

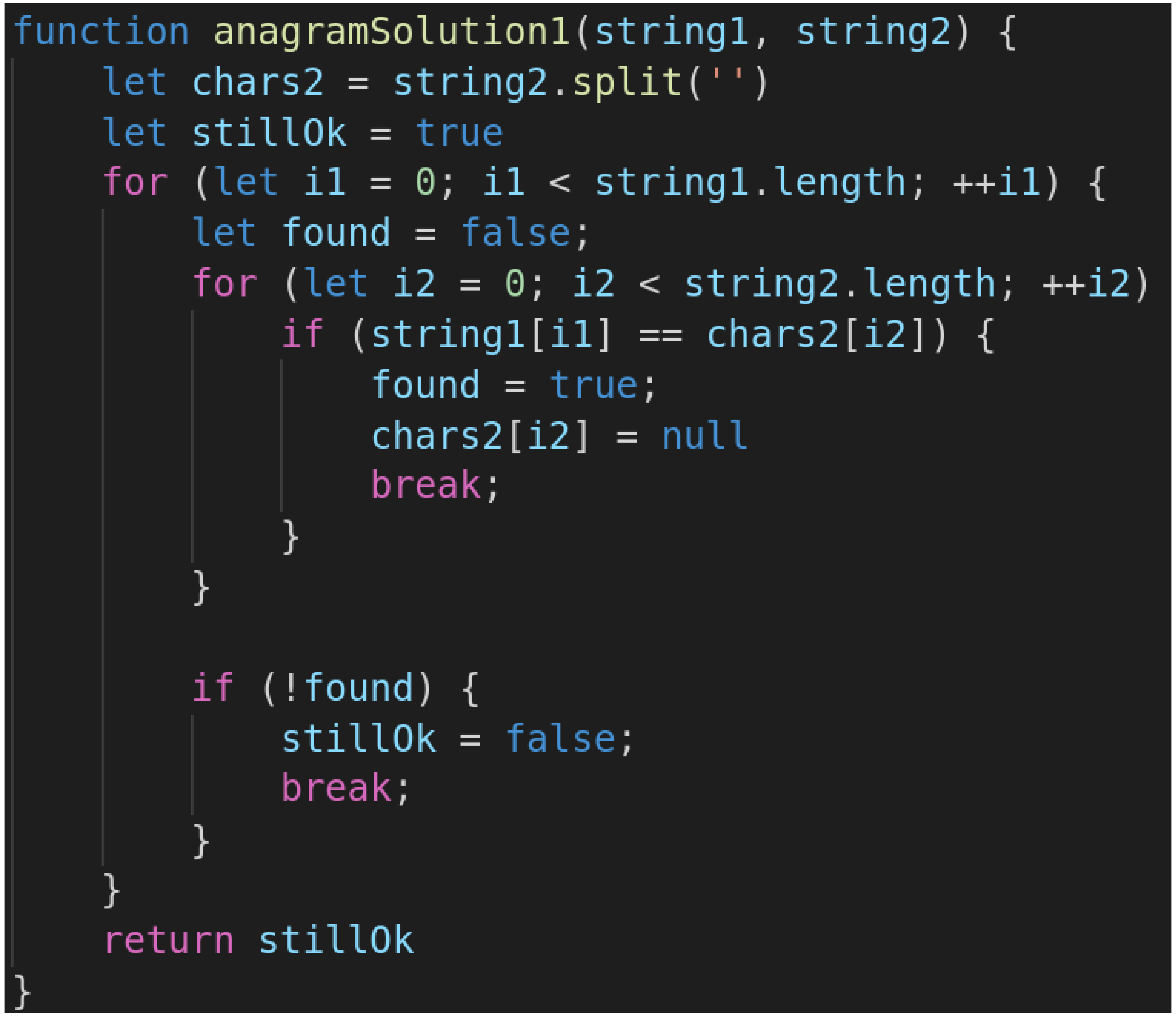
- $O(c\, f(x)) = O(f(x))$
- $O(f(x) + g(x)) = O(\max(f(x), g(x)))$
- $O(f(x)\, h(x)) < O(g(x)\, h(x))$ if and only if $O(f(x)) < O(g(x))$

Examples

- $O(4n) = O(n)$
- $O(n + \log n) = O(n)$
- $O(n \log n) < O(n^2)$ because $\log n < n$

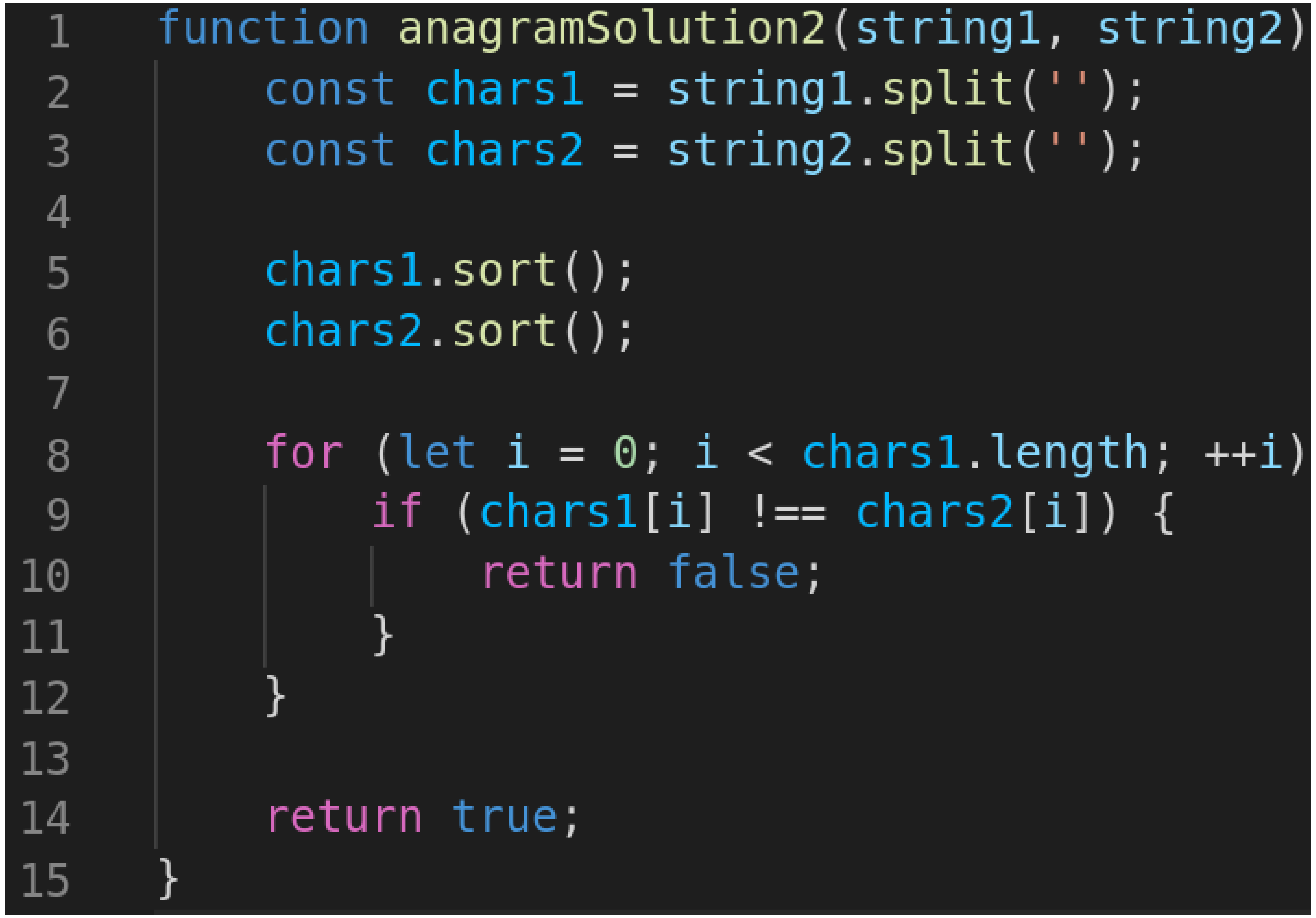
An Anagram Detection Example

A good example problem for showing algorithms with different orders of magnitude is the classic anagram detection problem for strings. One string is an anagram of another if the second is simply a rearrangement of the first. For example, 'heart' and 'earth' are anagrams. The strings 'python' and 'typhon' are anagrams as well. For the sake of simplicity, we will assume that the two strings in question are of equal length and that they are made up of symbols from the set of 26 lowercase alphabetic characters. Our goal is to write a boolean function that will take two strings and return whether they are anagrams.



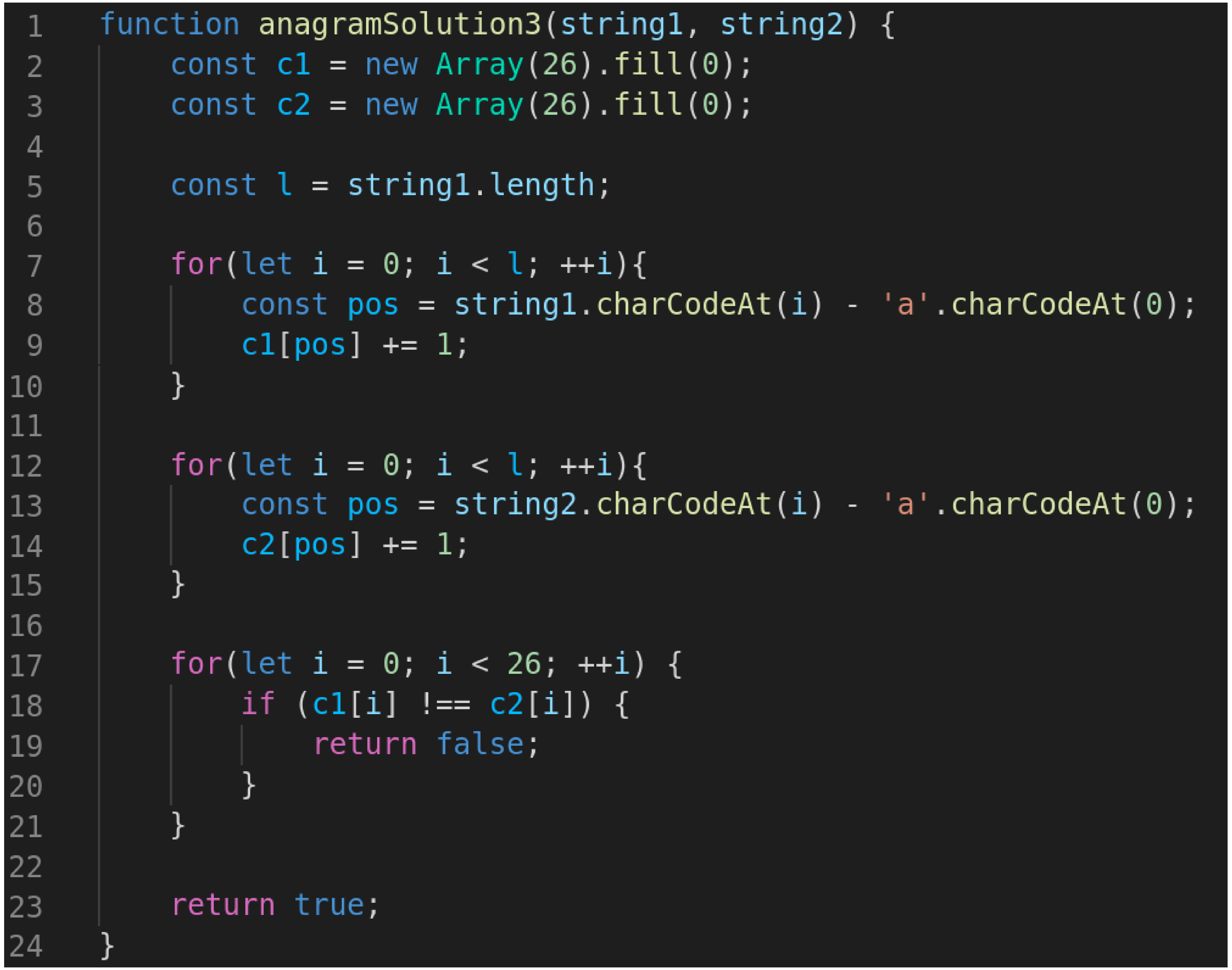
```
function anagramSolution1(string1, string2) {
    let chars2 = string2.split('')
    let stillOk = true
    for (let i1 = 0; i1 < string1.length; ++i1) {
        let found = false;
        for (let i2 = 0; i2 < string2.length; ++i2)
            if (string1[i1] == chars2[i2]) {
                found = true;
                chars2[i2] = null
                break;
            }
        }

        if (!found) {
            stillOk = false;
            break;
        }
    }
    return stillOk
}
```

O(n^2)



```
function anagramSolution2(string1, string2)
    const chars1 = string1.split('');
    const chars2 = string2.split('');

    chars1.sort();
    chars2.sort();

    for (let i = 0; i < chars1.length; ++i)
        if (chars1[i] !== chars2[i]) {
            return false;
        }
    }

    return true;
}
```

O(n logn)



```javascript
function anagramSolution3(string1, string2) {
    const c1 = new Array(26).fill(0);
    const c2 = new Array(26).fill(0);

    const l = string1.length;

    for(let i = 0; i < l; ++i){
        const pos = string1.charCodeAt(i) - 'a'.charCodeAt(0);
        c1[pos] += 1;
    }

    for(let i = 0; i < l; ++i){
        const pos = string2.charCodeAt(i) - 'a'.charCodeAt(0);
        c2[pos] += 1;
    }

    for(let i = 0; i < 26; ++i) {
        if (c1[i] !== c2[i]) {
            return false;
        }
    }

    return true;
}
```

O(n)