Big O Notation and complexity of Analysis for Algorithms

- Big O notation can objectively describe the efficiency of code without the use of concrete units
- Focus on how the time and space requirements scale
- prepare for the worst case scenario

sample.js

```
Const calculateAverage = (numbers) => {
    Let sum = 0;
    for (let i = 0; i < numbers.length; i++) {
        let number = numbers[i];
        sum += numbers;
    }
    return sum / numbers.length;
};
console.log(calculateAverage([2, 3, 4, 1])); // 2.5
```

there are n iterations in the for loop where n is the length of the array

$O(n)$ where n is the length of the input array

Simplifying Big O

product Rule

If the Big O is the product of multiple terms, drop the constant terms

forexample in ~~the prev~~ the example above there are 4 operations in the for loop hence $O(4 * n)$

When we drop the constant we get $O(n)$

Example 2.

$$O(512 * n) = O(n)$$

Example 3.

$$O(n/3) = O(1/3 \times n) = O(n)$$

Example 4

$O(5 * n * n) = O(n * n) = O(n^2)$

Example 5

$O(765) = O(1)$ This is also called constant time

Sum Rule

If the Big O is the sum of multiple terms, only keep the largest ~~sum~~ term, drop the rest

Sum Rule Example 1

$O(n + 1000)$

2 terms

1000 term is a constant

hence $O(n + 1000) = O(n)$

Sum Rule Example 2

$O(n^2 + n)$

2 terms

n^2 is the largest term

hence $O(n^2 + n) = O(n^2)$

Sum Rule Example 3

$O(n + 500 + n^3 + n^2)$

4 terms

n^3 is the largest term

hence $O(n + 500 + n^3 + n^2) = O(n^3)$

Putting it all together
To simplify fully, apply the product rule, followed by the sum rule

Full Simplification Example 1
$O(5n^2 + 100n + 17)$

Step 1: Apply product rule (drop constants)
$O(n^2 + n + 1)$

Step 2: Apply sum rule (Keep the largest term, drop the rest)
$O(n^2)$

Full simplification Example 2
$O((n/3)^6 + 10n)$
$O((1/3 * n)^6 + 10n)$

Step 1: product rule
$O(n^6 + n)$

Step 2: Sum rule
$O(n^6)$

Time Complexity Example 1

```
const foo = (n) => {
    for (let a = 0; a < n/2; a++){   O(n/2) = O(n)
        console.log(a);
    }
    for ( let b = 0; b < n; b++){    O(n)
        for ( let c = 0; c < n; c++){   O(n)
            console.log (b + "," + c);
        }
    }
};
foo (10);
```

the first loop is n
the second loop is $n * n = n^2$
$O(n + n^2) = O(n^2)$ where n is the input number

Time Complexity Example 2

```
const bar = (n) => {
    for (let i = 0; i < 3; i++) {            ...... 3
        for (let j = 0; j < n; j++) {        ...... n
            console.log (j);
        }
    }
    for (let k = 0; k < 10000; k++) {        ...... 10000
        console.log (k);
    }
};
bar (10);
```

first loop = 3n
second loop = 10000
$O(3n + 10000) = O(n)$ where n is the input number

Time Complexity Example 3

```
const boom = (n) => {
    for (let i = 0; i < 3; i++) {            ...... 3
        bam (n);                             ...... n
    }
    for (let k = 0; k < 10000; k++) {        ...... 10000
        console.log (k);
    }
};
```

```
const boom = (m) => {
    for (let j = 0; j < m; j++) {
        console.log(j);
    }
};
boom(10);
```

$O(3 * n + 10000) = O(3n + 10000) = O(n)$ where n is the input number

Space Complexity Example 1

```
const calculateAverage = (numbers) => {
    let sum = 0;  1
    for (let i = 0;  2  i < numbers.length; i++) {
        let number = numbers[i];  3
        sum += number;
    }
    return sum / numbers.length;
};
```

Space complexity $= O(3) = O(1)$

When software engineers refer to the term space complexity they are typically refering to any extra space that a solution may use not including the space consumed by the input array

Space Complexity Example 2

```
const doubler = (items) => {
    let newArray = [];
    for (let i = 0; i < items.length; i++) {   ...... n
        newArray.push(items[i]);               ...... 1
        newArray.push(items[i]);               ...... 1
    }
    return newArray;
};
doubler(['a', 'b', 'c']);
```

The for loop iterates through the array n times
Inside the for loop, we push the item into the newArray
twice
$O(n * (1+1)) = O(n*2) = O(2n) = O(n)$ where n is the
length of the input array

Analyzing Recursive Code

Our space complexity should consider the space taken by
recursive calls on the call stack

Recursive Example 1

```
const zoom = (n) => {
    if (n === 0) {
        console.log('liftoff!');
        return;
    }
    console.log(n);
    zoom(n-1);
};
zoom(10);
```

= $O(n)$ time, $O(n)$ space, where n is the input number

Recursive Example 2

```
const zap = (n) => {
    if (n < 1) {
        console.log('blastoff!');
        return;
    }
    console.log(n);
    zap(n-2);
};
zap(10);
```

Since we divide n by 2 compared to the previous example the $O(n/2) = O(1/2 \times n) = O(n)$

$O(n)$ time, $O(n)$ space, where n is the input number

Solving a problem

write a function, unique, that takes in an array and returns a new array containing the unique elements

Example:

unique(['cat', 'dog', 'rat', 'dog', 'cat', 'bird']);

should return: ['cat', 'dog', 'rat', 'bird']

~~Answer~~ Solution

```
const unique = (array) => {
    const newArray = [];
    for ( let i = 0; i < array.length; i++) { ........ n
        const ele = array[i];
        if (!newArray.includes(ele)) { ....... n
            newArray.push(ele);
        }
    }
    return newArray;
};
```

~~unique(['~~ time complexity $= O(n * n) = O(n^2)$

Space complexity is the space used up by our output array which is n

$= O(n^2)$ time, $O(n)$ space, where n is the input array size making the previous solution better

```
const unique = (array) => {
    const onlyUniques = new Set();
    for (let i = 0; i < array.length; i++) { ........ n
        const ele = array[i];
        onlyUniques.add(ele);
    }
    return Array.from(onlyUniques); ........ n
};
```

Time = $O(n+n) = O(2n) = O(n)$

Space = $O(n)$ where n is the input array size