# Estructuras de datos y algoritmos

## ¿Qué es buen código?

Es el código que tiene en cuenta:

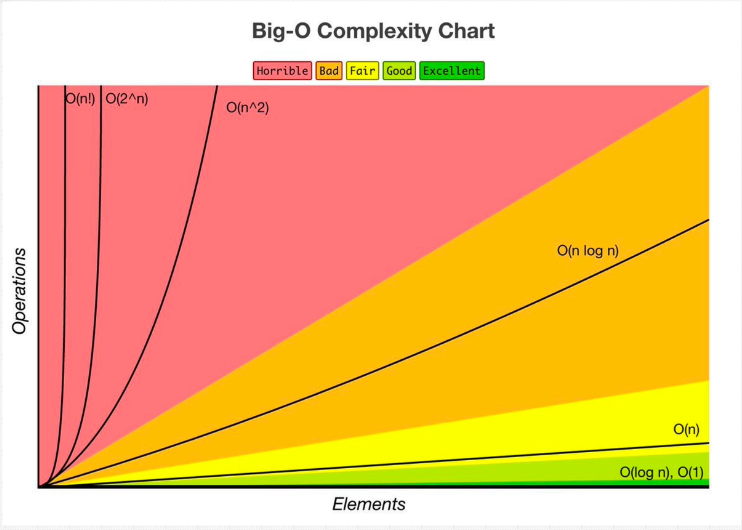
1. Legibilidad.
2. Veloz.
3. Memoria.

## ¿Que causa tiempo en una función?

1. Operaciones (+, -, \*, /)
2. Comparativas (<, >, =)
3. Looping (for, while)
4. Llamada a función externa (function())

## Big O

### Cuadro de complejidad Big O



### Reglas Big O

#### Regla 1: Peor caso

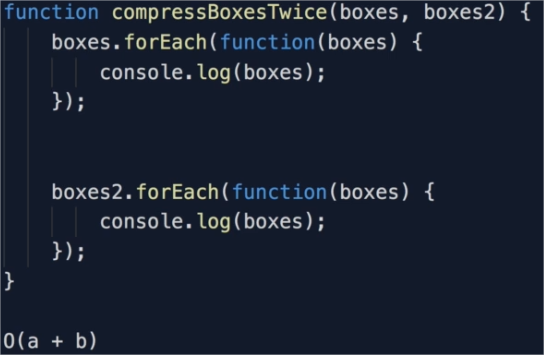
Siempre se calcula en base al peor caso.

#### Regla 2: Remueve las constantes

A la hora de expresar el resultado, no es necesario ser tan especifico, solo hay que mostrar el tipo de Big O que sería. Por ejemplo, tenemos O(n/2 + 101), esto se expresa como O(n). A esto se le dice remover las constantes.

#### Regla 3: Términos diferentes para inputs

Significa utilizar diferentes términos para diferentes inputs. Por ejemplo, teniendo en cuenta que boxes = a y boxes2 = b.



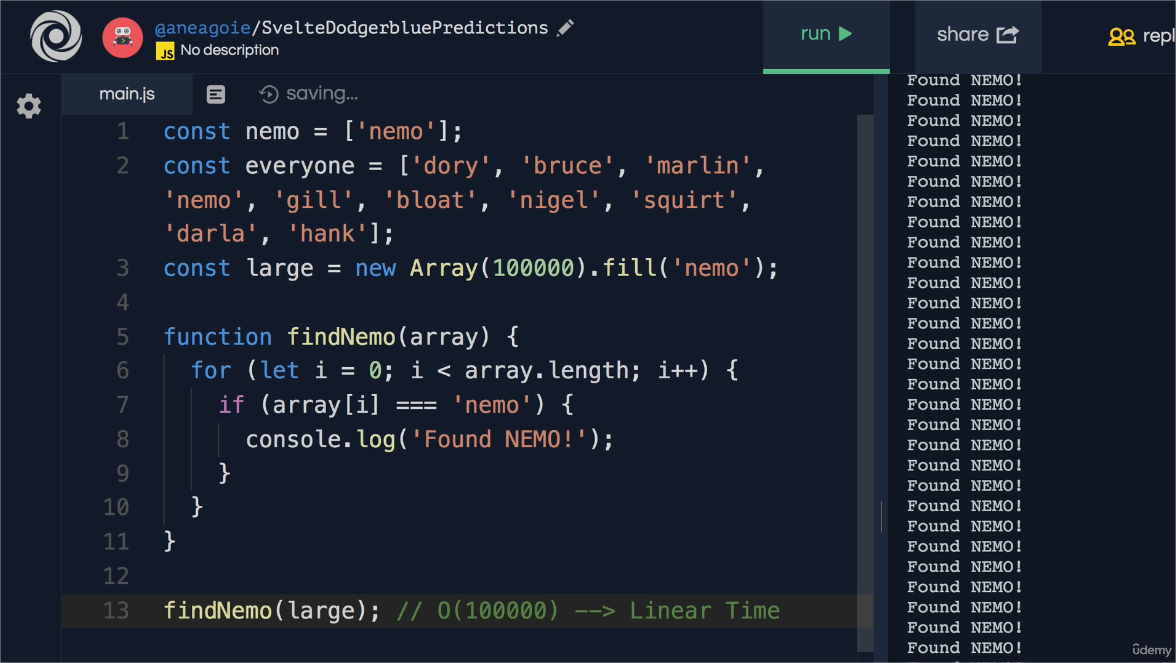
#### Regla 4: Elimina los no dominantes

Significa preocuparse por el termino más importante y eliminar el menos importante, es decir, se elimina el menos dominante.

Por ejemplo, cuando una función sea O(n + n^2), aplicando esta regla queda como O(n^2)

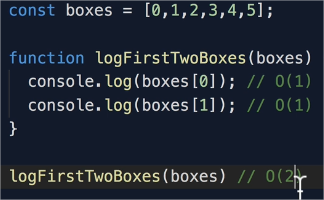
### O(n)

Es la notación linear/tiempo linear. Significa que el Big O depende de (n). Por ejemplo:



Es decir, el número de operaciones aumenta a medida que aumenta **n**.

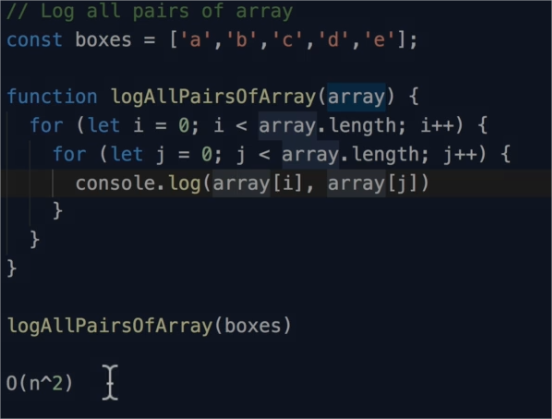
### O(1)

Es la notación constante/tiempo constante. Significa que siempre será una y solo una operación. Por ejemplo:  


A pesar de que se ejecuten dos operaciones y sea O(2), esto se describiría como O(1).

### O(n^2)

Es la notación cuadrática. Significa que el número de operaciones incrementa en proporción al cuadrado de n.



### O(n!)

Es la notación factorial. Normalmente no lo vas a encontrar, y si lo encuentras estás haciendo algo mal. Un ejemplo de O(n!) es un bucle anidado por cada entrada que recibe.

## Complejidad de espacio

Cuando un programa se ejecuta, tiene dos formas de recordar cosas:

Heap: Es donde se guardan las variables.

Stack: Es donde se hace el seguimiento de las funciones.

La complejidad de espacio (space complexity), es cuando se excede la capacidad de memoria, causando un Stack Overflow.

Lo que lo causan son:

* Variables
* Estructuras de datos
* Llamadas a funciones
* Alocaciones.

Ejemplo de calculo de complejidad:

//#5 Space complexity O(1)

function boooo(n) {

for (let i = 0 ; //O(1)

i < n; i++) {

console.log('booooo');

}

}

// #6 Space complexity O(n)

function arrayOfHiNTimes(n) {

var hiArray = [];

for (let i = 0; i < n; i++) {

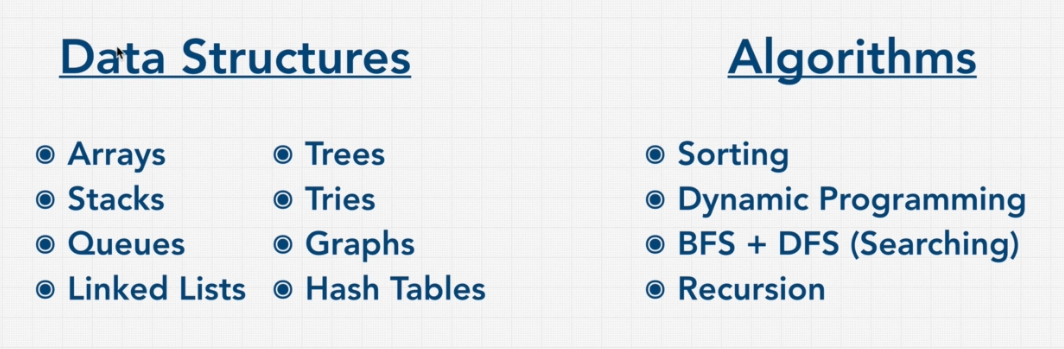
hiArray[i] = 'hi'; //O(n)

}

return hiArray;

}

## Estructuras de datos y algoritmos



### Arrays

Los arrays (también llamados listas), organizan elementos secuencialmente en memoria. Son los mejores para recorrer/iterar.

Lookup/Access: O(1)

Push: O(1)

Insert: O(n)

Delete: O(n)

Hay dos tipos de array:

* Estático: Tienen el tamaño fijo.
* Dinámico: Nos permite copiar y mover el array a otra localización en memoria, proveyendo más memoria. El Append/Push puede pasar a ser O(n).

### Hash tables

También llamados hash maps, maps, unordered maps, dictionaries, objects, etc., guardan en memoria un valor en base a una llave dada.

Insert: O(1)

Lookup: O(1)

Delete: O(1)

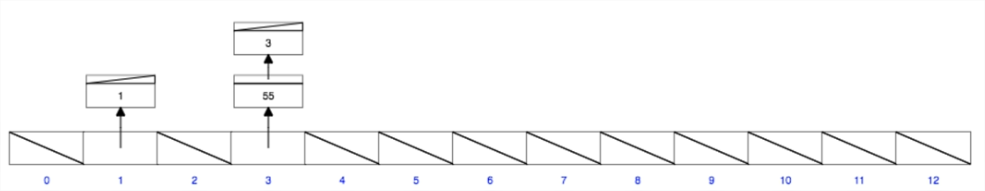
Search: O(1)

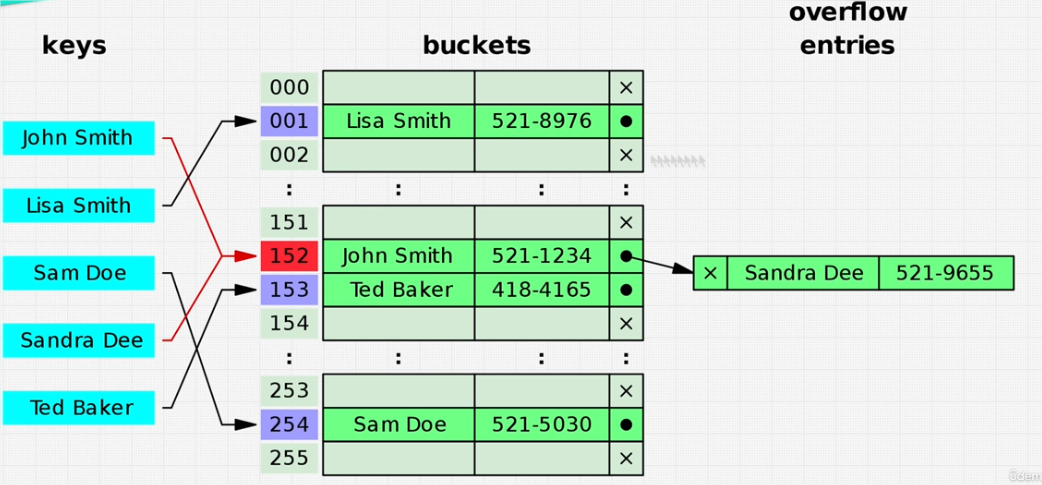
#### Colisión Hash

La principal contra de usar hash, es la colisión en memoria.

No hay nada que le diga a la función hash que el espacio en memoria está ocupado.

Esto hace que se forme una especie de lista en el espacio de memoria y aumenta la complejidad de tiempo a la hora de acceder/insertar/etc de O(1) a O(n), por ejemplo.





Ese “punto negro” que referencia a Sandra Dee, es lo que se forma una vez que hay una colisión y es una estructura de datos llamada “Linked List”, vista próximamente.