M1 - TEORIA Y REPASO

CLOSURE

Una clausura permite acceder al ámbito de una función exterior desde una función interior. En **JavaScript**, las clausuras se crean cada vez que una función es creada.

```
function saludar (saludo) {
  return function(nombre) {
      console.log(saludo + " " + nombre);
  }
}
```

RECURSION

Es usado para dividir el código en pequeños fragmentos. Mas sencillos de manejar.

Lo que hacemos en cada pasada es crear un nuevo stack, con un nuevo valor que luego se devolverá.

Dos cosas son necesarias para la recursión: caso de corte, donde se detendrá, y la recursión en sí.

```
function sumR(num) {
  if (num === 1) return 1;
  return num + sumR(num - 1);
}

sumR(4);
```

El Flow de esta function seria:

4 + sumR(3) 4 - 1 = 3

3 + sumR(2) 3 - 1 = 2

2 + sumR(1) 2 - 1 = 1

return 1

2 + sumR(1) --> 1 + 2 = 3

3 + sumR(2) --> 3 + 3 = 6

4 + sumR(3) --> 6 + 4 = 10

TOTAL = 10

Lo que esta función hace es sumar todos los números anterior al argumento que le paso, en este caso todos los números anterior a 4.

STACK

Es el modo en el que el motor de JS va ordenando todo lo que le llega. Similar a un "pila"(de ahí el "stack") de libro. Todo lo que le llega lo va ordenando arriba del anterior. Luego, para ejecutarlo, lo saca de arriba hacia abajo. De este modo la ultima instrucción que le llego será la primera en irse.

```
function takeShower() {
  return "Showering!";
}

function eatBreakfast() {
  let meal = cookFood();
  return `Eating ${meal}`;
}

function cookFood() {
  let items = ["Oatmeal", "Eggs", "Protein Sha
ke"];
  return items[Math.floor(Math.random() * item
s.length)];
}
```

```
function wakeUp() {
  takeShower();
  eatBreakfast();
  console.log("Ok ready to go to work!");
}

wakeUp();
```

Así funciona una stack:

- La función wakeUp contiene a su vez dos funciones más y un console.log.

- Cuando la llamamos esta función ejecutara primero takeShower y se quedara en pausa, no continúa ejecutando la siguiente función hasta que takeShower no se haya ejecutado.

- Ahora se ejecuta takeShower que retorna "Showering!". Termina y ahora volvemos a la funcion wakeUp.

- Se ejecuta eatBreakfast y se pone en pausa otra vez hasta que termina la ejecución.

- eatBreakfast a su vez llama a la función cookFood, por lo que se queda en pausa hasta que cookFood se termine de ejecutar.

- cookFood se ejecuta y devuelve un ítem aleatorio.

- Ahora eatBreakfast se puede terminar de ejecutar y devuelve `Eating "ítem aleatorio"`.

- Finalmente la función wakeUp se puede terminar de ejecutar y devuelve: "Ok ready to go to work!".

- Finish.

Big O notation

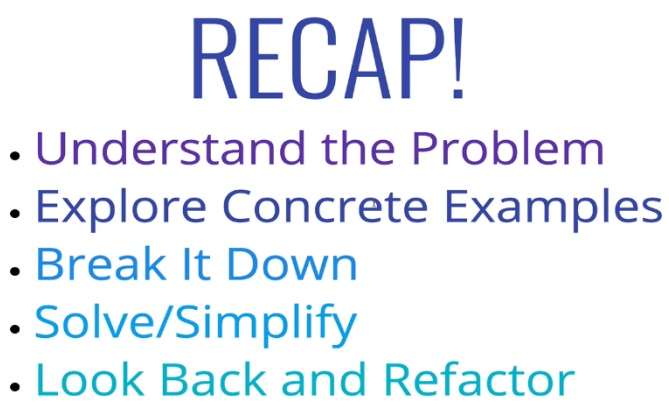
Es utilizado en ciencias computacionales para describir el rendimiento complejidad de un algoritmo. Generalmente describe el peor escenario, es decir, el máximo tiempo en la mayor cantidad de repeticiones que el algoritmo tiene que ejecutar.

```
Para los arrays la Big O Notation varia.

-- Cuando estamos tratando de acceder a un elemento

especifico del array lo hacemos mediante el índice
```

```
por lo que su valor será O(1).

-- Para insertar o eliminar un objeto depende.

Para .push() y .pop() estaremos hablando de O(1).

Sin embargo, para .shift() y .unshift() se usa O(n).

Ya que se debe re indexar todo el array cuando se usa.

¡SIEMPRE remover e insertar desde el final es más rápido!

-- Para la búsqueda se usa O(n).
```

Problem Solving

Como resolver un problema. Una buena alternativa es seguir estos cinco pasos:



Entender el problema.
Explorar ejemplos concretos.
Descomponerlos.
Resolver/ simplificar el problema.
Mirar hacia atrás y refactorizar.

Entender el problema

Para comenzar a entender el problema que se nos plantea deberíamos hacernos un par de preguntas, como:

* Puedo repetir el problema con mis propias palabras?

* Es importante poder entender el problema con nuestras propias palabras.

* Cuales son los problemas que plantea?

* Cual es el resultado que se espera?

* Se puede determinar el resultado a partir de la pregunta?

* ¿Es decir, tenemos suficientes informaciones para resolver el problema?

* Como debo etiquetar los datos importantes que son parte del problema?

* ¿Cuáles son las cosas importantes de este problema, como debo llamarlas, cual es la terminología?

==========

Escribe una función que tome dos números y retorne la suma de ambos.

==========

Inténtalo. Tu código aquí:

****

Explorar ejemplos concretos

* Comenzar con ejemplos simples.

* Ir agregando ejemplos cada vez más complicados.

* Explorar ejemplos con resultados vacíos.

* Explorar ejemplos con resultados inválidos.

==========

Escribe una función que tome una Sting y devuelva la cantidad de caracteres que posee.

==========

Inténtalo. Tu código aquí:

****

Descomponerlos

Escribir explícitamente los pasos que necesitas.

==========

Escribe una función que tome una string y devuelva la cantidad de caracteres que posee.

==========

Inténtalo. Tu código aquí:

****

Resolver/ simplificar el problema

En esta parte intentaremos simplificar el problema y resolver lo que podamos ignorando lo que más nos

está costando para más tarde. Lo cual se podría resumir:

* Encontrar lo que más te cuesta hacer.

* Ignóralo temporalmente.

* Escribe una solución simple.

* Ahora incorpora lo que más te estaba costando, si podes.

==========

Escribe una función que tome una string y devuelva la cantidad de caracteres que posee.

==========

Inténtalo. Tu código aquí:

****

Mirar hacia atrás y refactorizar

¡¡¡Todavía no terminamos!!! Antes de que podamos decir que el código está listo deberíamos hacernos estas preguntas:

* Podemos comprobar el resultado?

* Podemos llegar a ese resultado de manera diferente?

* Entendemos lo que hicimos?

* Podemos usar el código para resolver otro problema?

* Lo podemos mejorar?

* Podemos pensar otra forma de refactorizar nuestro código?

* Como lo resolvieron otras personas?

DATA STRUCTURES

Las estructuras de datos se pueden definir como cajas, o contenedores, donde se almacena

la información de acuerdo a ciertos parámetros. Por default JS ya contiene su propia Data Structure que son los objetos. Dentro de los objetos se pueden guardar cualquier tipo de datos, string, numbers, arrays, otros objetos.

Sin embargo a veces los objetos no son el mejor contenedor para guardar los datos. Por ende

veremos algunos tipos de Data Structures que podemos "manufacturar".

LINKED LIST

El más común y básico tipo de Data Structure.

Existen dos tipos de LL

- SINGLY LINKED LIST (SLL o LL)

- DOUBLY LINKED LIST (DLL)

Una Linked List es similar a un array en algunos aspectos, diferentes en otros.

Por ej: los dos permiten guardar una lista de valores casi infinita. Sin embargo

los datos dentro del array están indexados, esto quiere decir que cada elemento

tiene un índice asignado. En una LL los datos no están indexado, sino conectados,

enlazados unos con otros en forma de nodos.

En un array de, por ej, un millón de ítems podemos acceder fácilmente a un valor

solo con buscar su índice. Pero esta ventaja es también una desventaja. Si, en el

mismo ej, quisiéramos remover el primer elemento del array, podríamos hacerlo pero

los siguientes elementos deberían ser re indexados. Y esto cuesta tiempo y espacio.

Acá es donde las LL demuestran ser mejores, al tener sus datos conectados mediante

nodos y no por un índice las LL son mucho mejores para la adición y extracción de

datos que un array.

Otra fortaleza del array son sus métodos (push, pop, sort, etc). Las LL no poseen

métodos nativos por eso es necesario crearlos de forma “manual”.

Las LL poseen un nodo donde se guarda los parámetros de cómo van a ser almacenados. Poseen un valué o valor que recibirán, un next que apunta al siguiente valor y un prev que apunta al anterior valor, esto ultimo solo en DLL.
Luego la clase LL posee su propio constructor en donde está determinado la forma en que serán almacenados los datos. Este posee un head que será el principio, el primer dato que se almacene, un tail que será el último dato almacenado y un length que registrara la longitud de la LL, esto últimos es opcional.

En las SLL los datos se almacenan de la siguiente manera:
head --> val --> val --> val --> val --> val --> val --> tail

```
class Node{
    constructor(val){
        this.val = val;
        this.next = null;
    }
}

class SinglyLinkedList{
    constructor(){
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
    push(val){ // do something
    }
}
```

Solo pueden "mirar" hacia un solo sentido, hacia adelante.

Mientras que las DLL también pueden mirar hacia atrás:

head --> val --> val --> val --> val --> val --> val --> tail

<-- <-- <-- <-- <-- <-- <--

```
class Node{
    constructor(val){
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}

class DoublyLinkedList{
    constructor(){
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
}
```

Esto hace que las LL sean mucho más eficientes en cuanto a Big O Notation se refiere. Ya que pueden obtener o eliminar un dato almacenado sin tener que re indexar nada.

Pero, como dije, no poseen métodos "nativos" como los arrays. Por lo que hay que crearlos de modo "manual". Estos son algunos de ellos:

push, pop, shift, unshift, get, set, insert.

Como su nombre lo indica hacen lo mismo que los métodos de un array.

Push inserta un valor al final, pop lo remueve del final. Shift hace lo mismo

pero desde el principio, mientras que unshift remueve desde el principio, etc.

Existen varias "plantillas" sobre cómo organizar los datos dentro de una LL las más usadas son Queues y Stacks.

QUEUES

Es un modo de almacenar datos dentro de una LL regido bajo la norma FIFO. *First In, First Out*. El primer dato que llega será el primero en salir. Esto es un push y shift. Esto funcionaria así:

"First" --> "Second" --> "Thirt" --> "Fourt"

El primer valor que agregamos es "First". Los siguientes valores se irán agregando al final de la LL, es decir push. Pero cuando eliminemos un valor necesitamos que sea el primero en ser insertado para que se cumpla la norma FIFO, es decir hacemos un shift y extraemos "First".
Los métodos que utiliza un Queue para hacer esto son: enqueue y dequeue. Básicamente un push and unshift, respectivamente.

```
class Node {
  constructor(value){
      this.value = value;
      this.next = null;
  }
}

class Queue {
  constructor(){
      this.first = null;
      this.last = null;
      this.size = 0;
```

```
  }
  enqueue(val){

  }

  dequeue(){

  }
}
```

STACKS

En las stack se ordenan bajo la norma LIFO. Last In, First Out. Ultimo en entrar, primero en salir. Esto es un push and pop o también un shift and unshift.

"First" --> "Second" --> "Thirt" --> "Fourt"

El ultimo valor que agregamos es "Fourt", este será el primero en ser eliminado.
Los métodos que utiliza un Stack para hacer esto son: shift and unshift. Aunque técnicamente puede usar push and pop la idea detrás de una LL es que sea mucho mas efectiva y rápida que un array, de modo que si hacemos push and pop lo que hacemos es recorrer toda la lista e insertar/ remover el dato en cuestión. Esto quiere decir que sí tenemos un millón de objetos tendremos que atravesar ese millón de datos hasta llegar al final e inserta/ remover.
Con shift and unshift es mucho más sencillo ya que inserta/ remueve desde el principio y como no tiene índice no necesita re indexar nada. Por lo que es mucho más efectivo, rápido y bonito que un array y que un push and pop.

```
class Node {
  constructor(value){
      this.value = value;
```

```
        this.next = null;
    }
}

class Stack {
    constructor(){
        this.first = null;
        this.last = null;
        this.size = 0;
    }
    push(val){

    }
    pop(){

    }
}
```

TREE

La segunda estructura de dato que vamos a ver son los *TREEs*. Estos son estructuras no lineales que poseen nodos e hijos.

Existen un montón de tree pero nosotros vamos a enfocarnos en los *Binary Tree.*

BINARY TREE:

Se llaman así porque solo pueden poseer dos nodos por padre. Binario.

BINARY SEARCH TREE:

El ultimo tipo de Data Structure es lo que nos interesa, BST.

Al igual que LL es otra forma de almacenar datos solo que en vez de ser linear es

en forma de árbol con solo dos ramas, lo que le da su nombre. Esto es mucho mas

eficiente cuando necesitamos insertar o remover datos, más eficiente aun que las LL

ya que los BST utilizan O(LOG N). Son una versión mejorada del BT que posee todos los valores inferiores a su

izquierda y los mayores a la derecha. Lo cual lo hace ser super rápido para la búsqueda. Para ello contamos con dos métodos, además de otros, fundamentales para buscar: DFS y BFS.

```
class Node {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}

class BST {
  constructor() {
    this.root = null;
  }

  insert(value) {
    // do something...
  }

  find(value) {
    // do something...
  }
}
```

TREE TRAVERSAL

Cuando queremos buscar un elemento dentro de un BST es fácil, debido a que esta ordenado. Pero qué pasa cuando queremos buscar elementos dentro de un Tree, no un BST, que no están ordenados, ¿¿¿cómo hacemos???

Básicamente existen dos métodos populares para buscar a través de un Tree.
BREADTH FIRST SEARCH(BFS) y DEPTH FIRST SEARCH(DFS).

BREADTH FIRST SEARCH

Este método nos permite atravesar el árbol de forma horizontal.

```
        -->  10
  -->  6      -->    15
-->  3     8    -->       20
```

En este ejemplo, dado el árbol de arriba, con el método BFS si buscamos todos los elementos del árbol nos quedara lo siguiente: [10, 6, 15, 3, 8, 20].

DEPTH FIRST SEARCH

Por otro lado, permite buscar de forma vertical. Existen tres tipos de DFS: in order, pre order y post order.

HASH TABLE

Las hash tables implementan un tipo de dato abstracto asociada aun array que puede asignarles claves a los valores. Lo que, en otras palabras, es decir: las hash tables son colecciones de pares de datos key-value que trabajan con arrays.
Para ello las hash tables usan hash función para encontrar un índice o key, a menudo llamado hash code, dentro de los bukects que reciben los arrays. A partir de este hash code se puede encontrar el valor almacenado.

Las Colisiones ocurren cuando dos valores o más valores reciben el mismo índice. Para ello hay dos métodos que nos permiten manejar colisiones:
Separate Chaining y Linear Probing.

Separate Chaining: este método nos permite almacenar más de un valor en el mismo índice. De modo que cuando buscamos un valor no solo tendremos que comprobar el índice sino también que hay dentro y devolver el valor correcto.

Linear Probing: por otro lado, solo nos permite almacenar un valor por índice, de modo que si este está ocupado mira hacia el siguiente índice y pregunta si esta vacío. Sigue iterando así hasta encontrar un lugar vacío.

```
class HashTable {
  constructor() {
    this.numBackets = 35;
    this.buckets = [];
  }

  hash(key) {
    // do something...
  }
}
```

SEARCHING ALGORITHMS

Los algoritmos de búsqueda nos sirven para encontrar de la manera más optima y eficiente valores dentro de un array. Existen varios tipos de algoritmos, pero vamos a centrarnos en los 3 principales: *BUBBLE SORT, INSERTION SORT Y SELECTION SORT.*

BUBBLE SORT

Básicamente toma cada elemento del array y lo compara con el siguiente de acuerdo a lo que esperamos obtener, digamos que queremos ordenarlos de menor a mayor, toma el primer elemento y pregunta si es mayor al siguiente, si lo es los invierte, sino continua. El algoritmo se seguirá ejecutando mientras tenga números para ordenar o si después de una pasada completa no encontró ninguno.

> > >

[5, 2, 8, 1, 9] --> [2, 5, 8, 1, 9] --> [2, 5, 8, 1, 9] --> [2, 5, 1, 8, 9] ...

INSERTION SORT

Maquinita de peluche.

Este algoritmo compara el valor actual con el anterior y pregunta:

si el anterior es mayor que el actual saco el actual, como una máquina de peluches toma el premio, acá sacamos en una auxiliar el valor menor, el actual. Ahora esa posición será ocupada por el valor mayor y el menor, guardada en la auxiliar, ocupara la posición que el mayor tenía. Los invierte.

Toma el menor, como la garra de la maquinita, y lo coloca en la posición adecuada.

```
  aux              aux              aux
   |                |                |
   2                2
[5, 2, 8, 1, 9] -->  [5, 5, 8, 1, 9] -->   [2, 5, 8, 1, 9]
```

SELECTION SORT

Itera a través de todo el array buscando el valor mínimo. Cuando termina coloca ese valor al principio del array. Repita.

[5, 2, 8, 1, 9] --> [1, 2, 8, 5, 9]

PART 2

Los algoritmos de búsqueda que vimos hasta el momento cumplen su función pero son demasiado lentos.

Básicamente comparan todo el tiempo dos valores y los ordenan.

Vamos a ver dos algoritmos mucho más rápidos que mejoran la complejidad de O(N2) a O(N log N).

MERGE SORT

Merge, literalmente fusión, es una combinación de tres paso. Dividir, ordenar y fusionar. Recibe un

array. La divide en sub conjuntos de array hasta que queda múltiples array de un solo valor. Ordena

este array comprándolas y las fusiona.

Ejemplo:

[8,3,5,4,7,6,1,2] --> Dividimos hasta que cada valor se encuentre en un array.
[8,3,5,4] [7,6,1,2]
[8,3] [5,4] [7,6] [1,2]
[8] [3] [5] [4] [7] [6] [1] [2]
--- Ordenar y Fusionar. Compara cada subconjunto y lo ordena de menor a mayor ---
[3,8] [4,5] [6,7] [1,2]
[3,4,5,8] [1,2,6,7]
[1,2,3,4,5,6,7,8]

QUICK SORT

Para este algoritmo debemos tomar un numero al azar del array, llamado pivot, y ubicar todos los

valores menores a la izquierda y los mayores a la derecha. Esto no quiere decir que necesariamente

quedará ordenado, puede quedar de la siguiente forma:

[4,8,2,1,9,5,7,6,3] --> [1, 2, 3, 4, 8, 5, 7, 6, 9]

[26,23,27,44,17,47,39,42,43,1] --> [1, 23, 17, 26, 27, 47, 39, 42, 43, 44]

En estos dos ejemplos quick sort cumple su función ya que ordena todos los elementos menores a la

izquierda del pivot(4) y los mayores a la derecha. Aunque no estén ordenados cumple su función.

Ejemplo: [4,8,2,1,5,7,6,3]

[4, 8, 2, 1, 5, 7, 6, 3] --> Tomamos un pivot de forma aleatoria, por ejemplo: el indice 0, osea 4.

[3, 2, 1, 4, 5, 7, 6, 8] --> Ahora tenemos ubicado el 4 en su posicion final. Tenemos un elemento ordenado.

Ahora el pivot sera quien se encuentre en el indice 0, osea 3.

[1, 2, 3, 4, 5, 7, 6, 8] --> Ubicamos el 3 en su posicion final. Tomamos un nuevo pivot.

LINKS UTILES:

Clousers	https://www.youtube.com/watch?v=JXG_gQ0OF74&t=1450s
33 conceptos útiles	https://www.youtube.com/playlist?list=PLfWyZ8S-XzecAttp3QU-gBBXvMqEZTQXB
Pila de ejecución	https://developer.mozilla.org/es/docs/Glossary/Call_stack y https://www.youtube.com/watch?v=ygA5U7Wgsg8
Big O Notation	https://www.bigocheatsheet.com/
Searching algorithms	https://visualgo.net/es