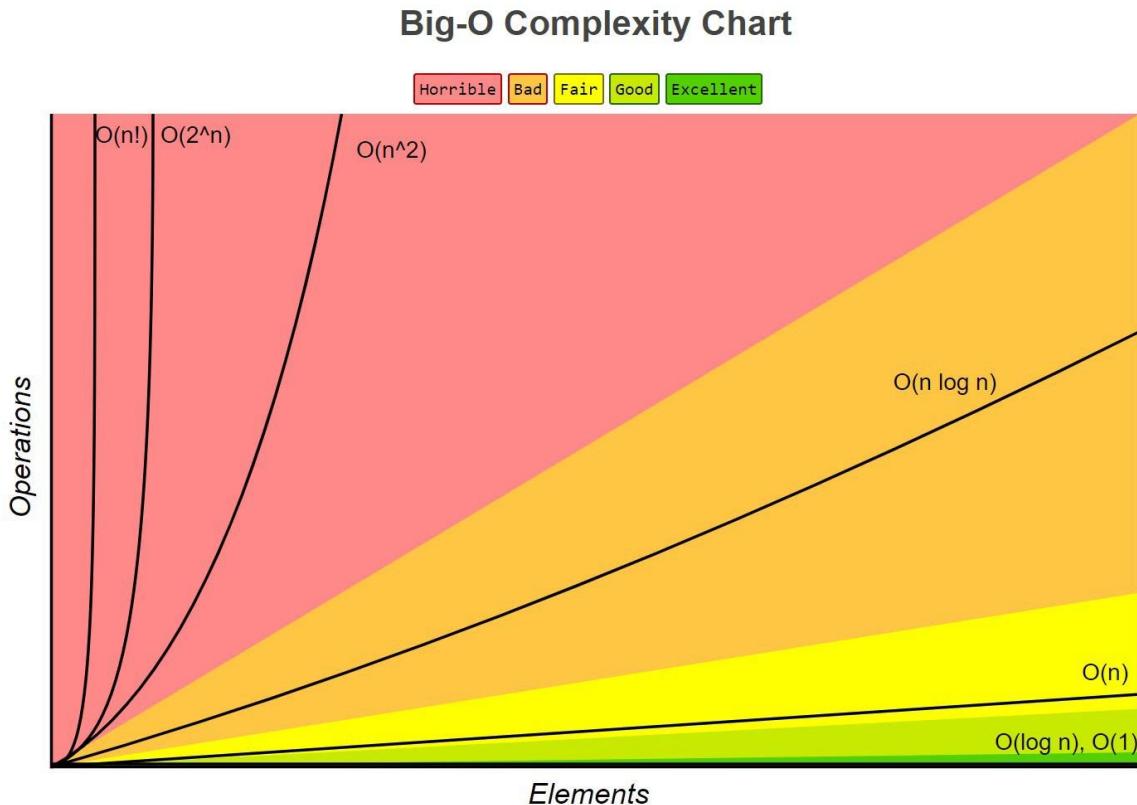


Algoritmos y Estructura de datos, utilizando JavaScript.

Para que todos podamos comprender la velocidad de un algoritmo, debemos primero aprender la notación matemática que nos permite identificar tanto el tiempo como el espacio de complejidad de los mismos.



También es importante seguir determinados pasos para poder solucionar los problemas que se nos presenten o que queramos intentar resolver:

Paso 1) Entender el problema.

Paso 2) Observar ejemplos concretos del mismo.

Paso 3) Romperlo en problemas más pequeños.

Paso 4) Resolverlo de manera simple.

Paso 5) Mirar atrás y refactorizar.

Patrones importantes para la solución de problemas:

Frequency Counter Pattern:

```
/*same, es una función que compara dos array
y debe determinar si tienen la misma cantidad de elementos sin importar el orden,
pero en el segundo array los elementos deben aparecer como los cuadrados del primero,
si esto se cumple retorna verdadero de lo contrario retorna falso.

ejemplo de verdadero:
[1,2,2,3,3,3] y [1,4,4,9,9,9] --> true

ejemplo de falso:
[1,2,3,2,5], [9,1,4,4,11] --> falso
*/
function same(arr1, arr2){
    if(arr1.length !== arr2.length){return false;} // compara la longitud de los array, retorna falso si son
diferentes.

    /* creamos un objeto que almacena como key un valor unico del array
    y como value la cantidad de veces que se repite ese mismo valor.*/
    let frequencyCounter1 = {}
    let frequencyCounter2 = {}
    //iteramos sobre los array y vamos almacenando las key y los values
    for(let val of arr1){
        frequencyCounter1[val] = (frequencyCounter1[val] || 0) + 1
    }
    for(let val of arr2){
        frequencyCounter2[val] = (frequencyCounter2[val] || 0) + 1
    }
    console.log(frequencyCounter1); // {"1": 1, "2": 2, "3": 1, "5": 1}
    console.log(frequencyCounter2); // {"1": 1, "4": 2, "9": 1, "11": 1}

    //comparamos los objetos
    for(let key in frequencyCounter1){
        if(!(key ** 2 in frequencyCounter2)){
            return false
        }
        if(frequencyCounter2[key ** 2] !== frequencyCounter1[key]){
            return false
        }
    }
    return true
}

same([1,2,3,2,5], [9,1,4,4,11]) //retorna falso
```

Multiple pointer pattern:

```

/*countUniqueValues, es una función que debe retornar la cantidad de valores distintos que tenemos en un array.
Ejemplo:
[1,2,2,5,7,7,99] --> retorna 5, porque tenemos 5 distintos valores 1,2,5,7,99
*/
function countUniqueValues(arr){
    if(arr.length === 0) return 0; // si la longitud del array es 0, directamente retornamos 0.
    var i = 0; // primer puntero es i
    //iteramos en el array
    for(var j = 1; j < arr.length; j++){ // segundo puntero es j, que se encuentra en la siguiente posición al puntero i.
        if(arr[i] !== arr[j]){ // si arr[i] es distinto a arr[j], aumentamos i en 1 y reasignamos el valor de arr[i], pero si son iguales, simplemente avanzamos +1 en j.
            i++;
            arr[i] = arr[j]
        }
    }
    return i + 1; //se retorna i + 1 porque i comenzó en 0.
}
countUniqueValues([1,2,2,5,7,7,99])

```

Sliding Window Pattern:

```

/*maxSubarraySum, es una función que debe retornar la suma máxima,
entre los elementos que indique el "num" con los elementos del array
Ejemplo:
[1,2,3,4], num = 1 --> retorna 4
[1,2,3,4], num = 2 --> retorna 3 + 4 = 7
[1,2,3,4], num = 3 --> retorna 2 + 3 + 4 = 9
[1,2,3,4], num = 4 --> retorna 1 + 2 + 3 + 4 = 10
*/
function maxSubarraySum(arr, num){
    let maxSum = 0; // variable para guardar la suma máxima
    let tempSum = 0; // suma temporal
    if (arr.length < num) return null; // si el número es mayor a la longitud del array, se devuelve nulo.
    //iteramos sobre el array
    for (let i = 0; i < num; i++) {
        //se inicializa la variable i = 0, se obtiene la suma con los primeros elementos hasta el valor num
        maxSum += arr[i]; // con el caso de [1,2,3,4] y el num = 3, esta iteración devuelve 1 + 2 + 3 = 6
    }
    tempSum = maxSum; // se almacena lo obtenido en la variable temporal
    //iteramos nuevamente
    for (let i = num; i < arr.length; i++) { // se inicializa i = num
        tempSum = tempSum - arr[i - num] + arr[i];
        //se aplica el patrón sliding window,
        //se resta el valor de la posición 0 y se le suma el siguiente
        [1,2,3,4] --> [1,2,3,4]
        [   ]     --> [   ]
        1+2+3     --> 2+3+4
    }
    maxSum = Math.max(maxSum, tempSum); // se decide cuál es mayor
}
return maxSum; // se retorna la suma máxima
}

maxSubarraySum([1,2,3,4],3)

```

Divide and Conquer Pattern:

```
/*countZeroes, es una función que debe retornar la cantidad de ceros, que se encuentran en el array,
este array se encuentra ordenado es decir primero aparecen todos los 1 y luego los 0
Ejemplo:
[1,1,1,1,1,0,0] ---> retorna 2
[1,0,0,0,0] ---> retorna 4
[1,1,1,1] ---> retorna 0
*/
function countZeroes(arr) {
    //se realiza una búsqueda binaria, es decir se divide al array en dos partes, las veces que sea necesaria.
    let left = 0; // variable para el extremo izquierdo del array
    let right = arr.length - 1; // variable para el extremo derecho del array
    while (left <= right) { //realizo un bucle
        let mid = Math.floor((left + right) / 2); //encuentro la mitad de mi array
        if (arr[mid] === 0 && (mid === 0 || arr[mid - 1] === 1)) {
            /*Si mi arr[mid]==0 y ademas (mid==0 o arr[mid-1] == 1),
            significa que arr[mid]== 0 es la primera aparición de un cero en mi array*/
            return arr.length - mid; // devuelvo directamente la longitud de mi array menos el valor de mi mitad
        } else if (arr[mid] === 1) {
            // Ahora si el valor arr[mid] === 1 significa que mis ceros se encuentran a la derecha de la mitad.
            left = mid + 1;
        } else {
            // Ahora si el valor arr[mid] === 0 solamente significa que puedo tener ceros previos al que acabo de
            encontrar.
            right = mid - 1;
        }
    }
    // Si no encuentro ceros en mi array, retorno directamente cero.
    return 0;
}
```

Recursividad: La recursividad consiste en funciones que se llaman a sí mismas, evitando el uso de bucles y otros iteradores. Es un método útil para ciertos tipos de problemas, ya que con una sola llamada nos permite recorrer toda una estructura. También entra en juego lo que denominamos “Call Stack” ya que un uso incorrecto de la recursividad, nos generará inconvenientes con ella.

```

/*collectOddValues, es una función que recibe un array
y retorna los números impares que hay en el mismo
Ejemplo:
[1,2,3,4,5] ---> retorna [1,3,5]
*/
function collectOddValues(arr){
    let newArr = []; //declaramos un nuevo array
    /*declaramos la condición base,
    esto es de suma importancia porque es la condición que va a detener
    el proceso recursivo.
    */
    if(arr.length === 0) {
        return newArr;
    }

    if(arr[0] % 2 !== 0){ //si la primera posición es impar pusheamos el valor a nuestro nuevo array
        newArr.push(arr[0]);
    }
    /* llamamos nuevamente a nuestra función,
    pero le realizamos un slice de 1, es decir eliminamos el primer elemento*/
    newArr = newArr.concat(collectOddValues(arr.slice(1)));
    return newArr;
}

collectOddValues([1,2,3,4,5])

```

Algoritmos de búsqueda

Linear Search: Time complexity O(n) en el peor de los casos. Dado que a medida que crece el array el número de iteraciones también crece.

```

/*linearSearch, es una función la cual toma como parámetros un array y un valor,
como resultado retorna el índice donde se encuentra ese valor y si no se encuentra en el array
retorna -1
Ejemplo:
[1,2,3,4,5], val = 3 ---> retorna 2
[1,2,3,4,5], val = 10 ---> retorna -1
*/
function linearSearch(arr, val){
    for(var i = 0; i < arr.length; i++){ //simplemente iteramos sobre el array una vez.
        if(arr[i] === val) return i;
    }
    return -1;
}

linearSearch([34,51,1,2,3,45,56,687], 100)

```

Binary Search: Time complexity $O(\log n)$ en el peor de los casos. Matemáticamente esta expresión sale de que la cantidad de pasos necesarias para encontrar el valor es $\log_2 N$, donde N es el tamaño del array.



```
/*binarySearch, es una función que recibe como parametro un array ordenado y un elemento a buscar,
al igual que linearSearch esta función retorna el índice o la posición del elemento dentro del array,
la diferencia es que binarySearch utiliza el patron divide and conquer para su solución.*/
function binarySearch(arr, elem) {
    var start = 0; //declaramos el extremo izquierdo
    var end = arr.length - 1; //declaramos el extremo derecho
    var middle = Math.floor((start + end) / 2); //obtenemos el índice que divide en dos al array.
    /*realizamos un bucle while
    Mientras el valor de la mitad en mi array sea diferente al elemento buscado
    y el extremo izquierdo menor al derecho el bucle continua.
    */
    while(arr[middle] !== elem && start <= end) {
        if(elem < arr[middle]) end = middle - 1; //reasignamos el valor del extremo derecho
        else start = middle + 1; //de lo contrario reasignamos el valor del extremo izquierdo
        middle = Math.floor((start + end) / 2); //volvemos a buscar el valor que se encuentra en la mitad
    }
    return arr[middle] === elem ? middle : -1; // finalmente retornamos el índice y si no se encontro
    retornamos -1
}

binarySearch([2,5,6,9,13,15,28,30], 28)

/*
[2,5,6,9,13,15,28,30]
[0           7]   ---> la mitad es 3
      3
      [4           7]   ---> como el valor buscado es 28, y la mitad era el valor 13, reasignamos el extremo
izquierdo a mid+1
      5           ---> es la mitad entre 4 y 7
      [6           7] ---> vuelve a reasignarse el extremo izquierdo
      6           ---> la mitad ahora, es el índice buscado, termina el bucle y se retorna el valor.
*/

```

Algoritmos de ordenación

Bubble Sort: es el algoritmo de clasificación más simple que funciona intercambiando repetidamente los elementos adyacentes si están en el orden incorrecto, mandando al final los valores más grandes, hasta que todos los valores se encuentren en su posición correspondiente. Time complexity $O(n^2)$ como promedio y en sus peores casos.

```

/*bubbleSort, es una función que tiene como parametro un array de números
y retorna el array ordenado, utilizando el algoritmo bubble Sort
Ejemplo:
[1,7,4,3,2,5,6] -> [1,2,3,4,5,6,7]
*/
function bubbleSort(arr){
    var noSwaps;
    /*declaramos la variable noSwaps que toma los valores true o false,
    dependiendo si realizamos intercambio entre posiciones de los elementos o no*/
    for(var i = arr.length; i > 0; i--){ // iteramos sobre el array, desde el final al principio. "primer
    puntero"
        noSwaps = true; // inicializamos la variable
        for(var j = 0; j < i - 1; j++){ // iteramos con la variable j, "segundo puntero"
            if(arr[j] > arr[j+1]){ // si el valor arr[j] es mayor que el valor siguiente a este realizamos el swap
                var temp = arr[j]; // guardamos temporalmente el valor arr[j]
                arr[j] = arr[j+1]; // asignamos en la posición de arr[j] el valor de arr[j+1]
                arr[j+1] = temp; // arr[j+1] le asignamos el valor de nuestra variable temp -> arr[j]
                noSwaps = false; // cambiamos a falso el valor de la variable noSwap
            }
        }
        if(noSwaps) break; // una vez que dejamos de iterar porque la condicion previa no se cumple mas, detenemos
        la iteración.
    }
    return arr; // retornamos el array ordenado
}

bubbleSort([8,1,2,3,4,5,6,7]);

```

Video: https://www.youtube.com/watch?v=nmhjrI-aW5o&ab_channel=GeeksforGeeks

Selection Sort: La ordenación por selección es un algoritmo de ordenación simple y eficiente que funciona seleccionando repetidamente el elemento más pequeño (o más grande) de la parte no ordenada de la lista y moviéndolo a la parte ordenada de la lista. Time complexity $O(n^2)$ como promedio y en sus peores casos.

```
/*selectionSort, es una función que recibe un array numérico y retorna el array de forma ordenada.  
Este proceso lo realiza mediante la implementación del algoritmo "selection sort",  
consiste en seleccionar repetidamente el elemento más pequeño (o más grande)  
de la parte no ordenada de la lista e ir moviéndolo a la parte ordenada de la lista.  
Ejemplo:  
[2,4,6,1,3,5] --> retorna [1,2,3,4,5,6]  
*/  
function selectionSort(arr) {  
    const swap = (arr, idx1, idx2) =>  
        ([arr[idx1], arr[idx2]] = [arr[idx2], arr[idx1]]); // arrow function, que realiza los swap entre los  
valores.  
    // iteramos sobre el array  
    for (let i = 0; i < arr.length; i++) {  
        let lowest = i; // declaramos la variable lowest, sera la encargada de tener el indice del numero  
seleccionado.  
        for (let j = i + 1; j < arr.length; j++) { // segunda iteración, con la variable j = i+1  
            if (arr[lowest] > arr[j]) {  
                lowest = j; //si la posición i > j, asignamos a lowest el valor de j  
            }  
        }  
        //por cada iteración, se pregunta si el valor de lowest cambio para realizar el swap.  
        if (i !== lowest) swap(arr, i, lowest);  
    }  
  
    return arr; // finalmente retornaos el array.  
}  
  
selectionSort([0,2,34,22,10,19,17]);
```

Video: https://www.youtube.com/watch?v=xWBP4lzkoyM&ab_channel=GeeksforGeeks

Insertion Sort: La clasificación por inserción es un algoritmo de clasificación simple que funciona de manera similar a la forma en que clasifica las cartas en sus manos. La matriz se divide virtualmente en una parte ordenada y otra no ordenada. Los valores de la parte no ordenada se seleccionan y colocan en la posición correcta en la parte ordenada. Time complexity $O(n^2)$ como promedio y en sus peores casos.

```

/*insertionSort, es una función que recibe un array numérico y retorna:
El mismo array pero de manera ordenada.
Para esta solución se implementa el algoritmo "insertion sort", consiste en que los valores
de la parte no ordenada se seleccionan y colocan en la posición correcta en la parte ordenada.
Ejemplo:
[4,7,3,1,8] --> retorna [1,3,4,7,8]
*/
function insertionSort(arr){
    var currentVal; // declaramos la variable current, para el manejo del valor de nuestro interés.
    // iremos declarando la variable i = 1
    for(var i = 1; i < arr.length; i++){
        currentVal = arr[i]; // le asignamos el valor correspondiente a current
        // segunda iteración con j = i-1
        for(var j = i - 1; j >= 0 && arr[j] > currentVal; j--) {
            /*Ejemplo de lo que sucede en la primera iteración, si el bucle cumple con la condición:
            [2,1,9,76,4]
            | |
            j i
            como j mayor a i, arr[j+1] que en este caso es i, se le asigna el valor de j.
        */
            arr[j+1] = arr[j]
        }
        /*Fuera del segundo bucle, es decir cuando ya se realizó j--,
        arr[j+1] ahora es la posición de j que vimos dentro del bucle, y a esta posición
        se le asigna el valor de currentVal que tenía guardado el valor de i inicial.
        ESTE PROCESO ES EL MISMO SWAP QUE VIMOS EN LOS OTROS MÉTODOS.
        */
        arr[j+1] = currentVal;
    }
    return arr; // finalmente retornamos el array
}

insertionSort([2,1,9,76,4])

```

Video: https://www.youtube.com/watch?v=OGzPmgsI-pQ&ab_channel=GeeksforGeeks

Merge Sort: se define como un algoritmo de ordenación que funciona dividiendo una matriz en subarreglos más pequeños, ordenando cada subarreglo y luego fusionando los subarreglos ordenados para formar el arreglo ordenado final. Time complexity $O(n \cdot \log(n))$.

```

/*merge, es una función a la cual se le pasa como argumento dos arrays numéricos ordenados, y retorna un solo array ordenado.
Ejemplo:
[2],[1] --> retorna [1,2] */

function merge(arr1, arr2){
    let results = []; //declaramos el nuevo array
    //inicializamos los punteros i y j
    let i = 0;
    let j = 0;
    //declaramos un bucle while, con las siguientes condiciones
    while(i < arr1.length && j < arr2.length){
        if(arr2[j] > arr1[i]){ //si arr2[j] es mayor a arr1[i], pusheamos el segundo valor y aumentamos su puntero en 1.
            results.push(arr1[i]);
            i++;
        } else {
            results.push(arr2[j]);
            j++;
        }
    }
    /*por ultimo declaramos dos bucles mas,
    los cuales pushean los valores del arr1 o arr2, si es que quedaron valores sin comparar.
    Ejemplo:
    [2,3], [1,4,5] --> [1,2,3] y en el arr2 quedan los valores [4,5],
    entonces los pusheamos dentro del array final [1,2,3,4,5]
    */
    while(i < arr1.length) { //
        results.push(arr1[i])
        i++;
    }
    while(j < arr2.length) {
        results.push(arr2[j])
        j++;
    }
    return results; //finalmente lo retornamos
}

// Merge Sort algoritmo, con recursividad.
function mergeSort(arr){
    if(arr.length <= 1) return arr; //caso base
    let mid = Math.floor(arr.length/2); //obtenemos la mitad del array
    /*volvemos a llamar a mergerSort, pero con el array modificado desde la posición inicial hasta la mitad,
    este proceso continua hasta que el array queda de solo un elemento, es decir sigue la estrategia de divide and conquer*/
    let left = mergeSort(arr.slice(0,mid));
    //misma lógica para la parte derecha del array
    let right = mergeSort(arr.slice(mid));
    return merge(left, right); //llamamos a la función merge, para obtener un array único ordenado.
}

mergeSort([10,24,76,73])

```

Video: https://www.youtube.com/watch?v=JSceec-wEyw&ab_channel=GeeksforGeeks

Quick Sort: en promedio, hace $O(n \cdot \log(n))$ comparaciones para ordenar n elementos. En el peor de los casos, hace $O(n^2)$ comparaciones, aunque este comportamiento es muy raro.

Funcionalidad general:

- Selección de pivote: Elija un elemento, llamado pivote, del array (generalmente el elemento más a la izquierda o más a la derecha de la partición).
- Fraccionamiento: Reordene el array de manera que todos los elementos con valores menores que el pivote estén antes del pivote. Por el contrario, todos los elementos con valores mayores que el pivote vienen después de este. Los valores iguales pueden ir en cualquier dirección. Después de esta partición, el pivote está en su posición final.
- Repetirse: Aplique recursivamente los pasos anteriores al subarray de elementos con valores más pequeños que el pivote y por separado al subarray de elementos con valores mayores que el pivote.

```

/*pivot, es una función que tiene como parametros un array desordenado, un valor que indica su comienzo y
otro que indica su final,
la misma ordenada el primer elemento del array en la posición que le corresponde y retorna su índice.
Ejemplo:
[100,-3,2,4,6,9,1,2,5,3,23]--> modifica el array [23,-3,2,4,6,9,1,2,5,3,100] --> retorna 10.
*/
function pivot(arr, start = 0, end = arr.length - 1) {
    const swap = (arr, idx1, idx2) => { //declaramos nuestra función swap
        [arr[idx1], arr[idx2]] = [arr[idx2], arr[idx1]];
    };

    let pivot = arr[start]; //declaramos nuestro pivot, que es el primer elemento del array.
    let swapIdx = start; //guardamos el valor de la variable start, dentro de swapIdx.

    /*Declaramos un bucle for, que inicializa en la posición siguiente a start,
    comparamos el pivot con cada uno de los elementos del array, si se cumple que el pivot es mas grande,
    realizamos swapIdx++*/
    for (let i = start + 1; i <= end; i++) {
        if (pivot > arr[i]) {
            swapIdx++;
        }
    }
    // Por ultimo realizamos el swap, entre el elento de inicio y el pivot.
    swap(arr, start, swapIdx);
    return swapIdx; //retornamos la posicion final del pivot.
}

//quickSort algoritmo y recursividad.
function quickSort(arr, left = 0, right = arr.length -1){
    if(left < right){ //siempre que el extremo izquierdo sea menor al extremo derecho.
        //obtenemos el indice del pivot y a su vez ordenamos en la posición que corresponde al elemento inicial.
        let pivotIndex = pivot(arr, left, right)
        // volvemos a llamar a la función quickSort pero ahora para los valores de la izquierda en referencia al
        //pivot.
        quickSort(arr,left,pivotIndex-1);
        // volvemos a llamar a la función quickSort pero ahora para los valores de la derecha en referencia al pivot.
        quickSort(arr,pivotIndex+1,right);
    }
    return arr;
}

quickSort([100,-3,2,4,6,9,1,2,5,3,23])

```

Video: https://www.youtube.com/watch?v=PgBzjlCcFvc&ab_channel=GeeksforGeeks

Dato: El límite inferior para el algoritmo de clasificación basado en comparación (Merge Sort, Quick Sort, etc) es $\Omega(n \log n)$, es decir, no pueden hacerlo mejor que $n \log n$.

El algoritmo Radix Sort, propone otro método, utilizando una propiedad de los números naturales para poder resolverlo de manera más eficiente a ciertos casos en específico:

La idea de Radix Sort es ordenar dígito por dígito comenzando desde el dígito menos significativo hasta el dígito más significativo. Radix sort usa la ordenación por conteo como una subrutina para ordenar. Su time complexity es de $O(n.k)$

```

● ● ●

//La función getDigit, se encarga de obtener el dígito en la posición i del número num.
function getDigit(num, i) {
    return Math.floor(Math.abs(num) / Math.pow(10, i)) % 10;
}
//verifica la cantidad de dígitos que tiene un número en particular, luego se implementa esta función en
mostDigits.
function digitCount(num) {
    if (num === 0) return 1;
    return Math.floor(Math.log10(Math.abs(num))) + 1;
}
// es una función que itera al array de números y devuelve aquel que tiene mayor cantidad de dígitos.
function mostDigits(nums) {
    let maxDigits = 0;
    for (let i = 0; i < nums.length; i++) {
        maxDigits = Math.max(maxDigits, digitCount(nums[i]));
    }
    return maxDigits;
}
/*
La función radixSort, utiliza el algoritmo de ordenamiento Radix Sort para ordenar el array.
Primero se obtiene la cantidad de dígitos del número con más dígitos en nums utilizando la función
mostDigits(nums).
Luego, se itera sobre cada posición de los dígitos, desde la posición menos significativa a la más
significativa,
utilizando un bucle for que va de 0 a maxDigitCount - 1.
En cada iteración se crea un array digitBuckets con 10 elementos vacíos, que representan los posibles valores
de un dígito (0 a 9).
Luego, se itera sobre cada número de nums, se obtiene el dígito correspondiente a la posición actual utilizando
la función getDigit(num, i)
y se inserta el número en el elemento correspondiente del array digitBuckets.
Finalmente, se concatena el contenido de todos los elementos de digitBuckets en un nuevo array nums.
*/
function radixSort(nums){
    let maxDigitCount = mostDigits(nums);
    for(let k = 0; k < maxDigitCount; k++){
        let digitBuckets = Array.from({length: 10}, () => []);
        for(let i = 0; i < nums.length; i++){
            let digit = getDigit(nums[i],k);
            digitBuckets[digit].push(nums[i]);
        }
        nums = [].concat(...digitBuckets);
    }
    return nums;
}

radixSort([23,345,5467,12,2345,9852])

```

Video: https://www.youtube.com/watch?v=nu4gDuFabIM&ab_channel=GeeksforGeeks

Estructuras de datos:

Singly Linked List: Una lista enlazada individualmente es una estructura de datos lineal en la que los elementos no se almacenan en ubicaciones de memoria contiguas y cada elemento se conecta solo a su siguiente elemento mediante un puntero. Time complexity en comparación al array tradicional:

Operation	linked list	Array
Random access	$O(n)$	$O(1)$
Insertion and Deletion at the beginning	$O(1)$	$O(n)$
Insertion and Deletion at the end	$O(n)$	$O(1)$
Insertion and Deletion from random location	$O(n)$	$O(n)$

Time Complexity (Big O) Comparisons

Estructura principal:

```
//NODO, CONTIENE UN VALOR Y UN PUNTERO QUE INDICA EL SIGUIENTE NODO
class Node{
    constructor(val){
        this.val = val;
        this.next = null;
    }
}
// SIMPLY LINKED LIST, TIENE UNA CABEZA, UNA COLA Y UNA LONGITUD, INICIALIZADAS EN NULO Y CERO.
class SinglyLinkedList{
    constructor(){
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
}
```

Métodos de las listas enlazadas individualmente:

```

//METODO PUSH, AGREGA UN NODO A LA LISTA.
push(val){
    var newNode = new Node(val); //se crea un nuevo nodo
    //en este punto se verifica si existe una cabeza, tambien podemos evaluar la longitud de la lista.
    if(!this.head){
        this.head = newNode;
        this.tail = this.head;
    }
    /*Si ya existe una cabeza, se agrega el nodo al final de la lista, indicando que la cola ahora apunta a un
    nuevo valor,
    y se reasigna la cola al nuevo valor agregado.
    */
    else {
        this.tail.next = newNode;
        this.tail = newNode;
    }
    this.length++; //luego de realizar el push, se incrementa la longitud en 1.
    return this;
}
//ESTE METODO ELIMINA EL NODO QUE SE ENCUENTRA AL FINAL DE LA LISTA.
pop(){
    if(!this.head) return undefined; //evalua que la lista no este vacia.
    var current = this.head;
    var newTail = current;
    while(current.next){ //bucle para ir avanzando dentro de la lista.
        newTail = current; //valor previo, que se va a convertir en cola.
        current = current.next; //valor al final de la lista.
    }
    this.tail = newTail; //reasignamos la cola con el valor newTail.
    this.tail.next = null; //eliminados el nodo que se encontraba al final de la lista.
    this.length--; //reducimos en 1 la longitud de la lista.
    //si la longitud de la lista era 1, comprobamos si llegamos a una longitud de cero, y eliminamos tambien la
    //cabeza de la lista.
    if(this.length === 0){
        this.head = null;
        this.tail = null;
    }
    return current; //retornamos el valor que eliminamos.
}
//ELIMINA EL PRIMER NODO DE LA LISTA.
shift(){
    if(!this.head) return undefined; //comprobamos si la lista esta vacia.
    var currentHead = this.head; //guardamos la cabeza de la lista en la variable currentHead
    this.head = currentHead.next; //reasignamos la cabeza de la lista, al valor siguiente.
    this.length--; //reducimos la longitud en 1.
    if(this.length === 0){ //si la longitud previa era 1 y ahora se convierte en 0, debemos hacer nula la
    //cola tambien.
        this.tail = null;
    }
    return currentHead; //retornamos el valor eliminado.
}
//AGREGA UN NODO AL INICIO DE LA LISTA.
unshift(val){
    var newNode = new Node(val); //creamos un nuevo nodo
    if(!this.head) { //comprobamos la longitud o si existe la cabeza de la lista.
        this.head = newNode;
        this.tail = this.head;
    }
    newNode.next = this.head; //indicamos a donde va a punchar el nuevo nodo.
    this.head = newNode; //reasignamos el valor de la cabeza al nuevo nodo.
    this.length++; //aumentamos la longitud en 1.
    return this; //retornamos la lista.
}

```

Métodos para obtener, editar, insertar o remover cualquier valor de lista:

```

//OBTIENE EL NODO DEL INDEX ESPECIFICADO.
get(index){
    //si el índice es menor a 0 o mayor a la longitud de la lista, retornaos null.
    if(index < 0 || index >= this.length) return null;
    var counter = 0; //declaramos la variable counter.
    var current = this.head; //guardamos el nodo cabeza de la lista en la variable current.
    while(counter !== index){ //realizamos un bucle
        current = current.next; //avanzamos en la lista
        counter++; //aumentamos el valor de counter
    }
    return current; //retornamos el nodo buscado.
}
// REASIGNA EL VALOR DEL NODO INDICADO.
set(index, val){
    var foundNode = this.get(index); //llamamos a la función get para obtener el nodo indicado.
    if(foundNode){ //si existe el nodo buscado
        foundNode.val = val; //reasignamos su valor
        return true; //retornamos true
    }
    return false; //si el nodo buscado no existe, retornamos false.
}
//INSERTA UN NODO EN CUALQUIER PARTE DE LA LISTA.
insert(index, val){
    if(index < 0 || index > this.length) return false; //comprobamos que el índice sea un valor valido en
    la lista.
    //si el indice es igual a la longitud de la lista, simplemente pusheamos el nuevo nodo.
    if(index === this.length) return !!this.push(val);
    //si el inddice es igual a cero, llamamos a la función unshift para agregarlo al inicio.
    if(index === 0) return !!this.unshift(val);

    //de lo contrario, declaramos el nuevo nodo
    var newNode = new Node(val);
    var prev = this.get(index - 1); //obtenemos el nodo previo al deseado.
    var temp = prev.next; //guardamos el puntero que nos indica a donde apunta el nodo previo con la
    variable temp.
    prev.next = newNode; //le indicamos al nodo previo que ahora apunta al nuevo nodo.
    newNode.next = temp; // al nuevo nodo le indicamos que debe apuntar al nodo que ya existia en su
    posicion.
    this.length++; //aumentamos la longitud en 1.
    return true; //retornamos verdadero.
}
//REMUEVE UN NODO EN CUALQUIER PARTE DE LA LISTA.
remove(index){
    if(index < 0 || index >= this.length) return undefined;//comprobamos que el índice sea un valor
    valido en la lista.
    if(index === 0) return this.shift(); //si es igual a 0, eliminamos el nodo del principio.
    if(index === this.length - 1) return this.pop(); //si es igual a la longitud - 1, eliminamos el nodo
    del final.
    var previousNode = this.get(index - 1); //buscamos el nodo previo, llamando a get.
    var removed = previousNode.next; //guardamos el nodo que vamos eliminar en una variable.
    //le indicamos al nodo previo, que ahora apunte al siguiente nodo que se encuentra despues del nodo a
    eliminar.
    previousNode.next = removed.next;
    this.length--; //reducimos la longitud en 1.
    return removed; //devolvemos el nodo eliminado.
}

```

Método para invertir e imprimir en forma de array la lista:

```

//INVIERTE LA DIRECCION DE LOS NODOS, "CABEZA ---> COLA" SE TRANSFORMA EN "COLA ---> CABEZA"
reverse(){
    //En primer lugar, se guarda el nodo que actualmente está en la cabeza de la lista en la variable node.
    var node = this.head;
    /*se intercambia el valor de la cabeza de la lista con el de la cola de la lista,
    mediante la asignación this.head = this.tail y this.tail = node. */
    this.head = this.tail;
    this.tail = node;
    /*se declaran dos variables: next para guardar el nodo siguiente al nodo actual en cada iteración,
    y prev para guardar el nodo anterior al nodo actual */
    var next;
    var prev = null;
    for(var i = 0; i < this.length; i++){ //recorremos la lista con un bucle for.
        /*
        En cada iteración, se guarda el valor del nodo siguiente a node en la variable next.
        Luego se asigna el valor de prev al nodo siguiente a node mediante la expresión node.next = prev,
        con lo cual se invierte el orden del enlace.Por último, se actualizan los valores de prev y node para la
        siguiente iteración.
        */
        next = node.next;
        node.next = prev;
        prev = node;
        node = next;
    }
    return this;
}
//DEVUELVE LOS VALORES DE LOS NODOS QUE CONTIENE LA LISTA EN FORMA DE ARRAY.
print(){
    var arr = []; //declaramos el array
    var current = this.head // guardamos el valor de la cabeza de la lista en la variable current.
    while(current){ //iteramos
        arr.push(current.val) //pusheamos el valor del nodo current dentro del array.
        current = current.next //vamos atravesando toda la lista.
    }
    console.log(arr); //mostramos el array en la consola.
}

```

Crear un objeto con la estructura de la lista:

```
var list = new SinglyLinkedList()
```

Doubly Linked List: Una lista doblemente enlazada (DLL) es un tipo especial de lista enlazada en la que cada nodo contiene un puntero al nodo anterior y al siguiente nodo de la lista enlazada. Time Complexity:

Operation	Singly Linked List	Doubly Linked List
insert at head	O(1)	O(1)
insert at tail	O(1)	O(1)
remove at head	O(1)	O(1)
remove at tail	O(n)	O(1)
remove in middle	O(n)	O(n)
seara	O(n)	O(n)
get from index	O(n)	O(n)

Estructura principal:

```

● ● ●

// La clase Node agrega el parametro prev, para poder apuntar tanto al siguiente nodo como al anterior.
class Node{
    constructor(val){
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}

//Misma estructura que la Singly Linked List.
class DoublyLinkedList {
    constructor(){
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
}

```

Métodos de las listas dobles:

```
//Añadir un valor al final
push(val){
    var newNode = new Node(val);
    if(this.length === 0){
        this.head = newNode;
        this.tail = newNode;
    } else {
        this.tail.next = newNode;
        newNode.prev = this.tail;
        this.tail = newNode;
    }
    this.length++;
    return this;
}
//Quitar un valor del final.
pop(){
    if(!this.head) return undefined;
    var poppedNode = this.tail;
    if(this.length === 1){
        this.head = null;
        this.tail = null;
    } else {
        this.tail = poppedNode.prev;
        this.tail.next = null;
        poppedNode.prev = null;
    }
    this.length--;
    return poppedNode;
}
//Quitar un valor del principio.
shift(){
    if(this.length === 0) return undefined;
    var oldHead = this.head;
    if(this.length === 1){
        this.head = null;
        this.tail = null;
    }else{
        this.head = oldHead.next;
        this.head.prev = null;
        oldHead.next = null;
    }
    this.length--;
    return oldHead;
}
//Añadir un valor al principio.
unshift(val){
    var newNode = new Node(val);
    if(this.length === 0) {
        this.head = newNode;
        this.tail = newNode;
    } else {
        this.head.prev = newNode;
        newNode.next = this.head;
        this.head = newNode;
    }
    this.length++;
    return this;
}
```

Obtener y Editar un nodo, métodos:

```
//Obtener un nodo mediante su indice.
get(index){
    if(index < 0 || index >= this.length) return null;
    var count, current;
    if(index <= this.length/2){
        count = 0;
        current = this.head;
        while(count !== index){
            current = current.next;
            count++;
        }
    } else {
        count = this.length - 1;
        current = this.tail;
        while(count !== index){
            current = current.prev;
            count--;
        }
    }
    return current;
}
//Asignar un valor al nodo indicado.
set(index, val){
    var foundNode = this.get(index);
    if(foundNode != null){
        foundNode.val = val;
        return true;
    }
    return false;
}
```

Insertar o Eliminar un nodo en cualquier posición, método:

```
//Insertar un nuevo nodo en la lista.
insert(index, val){
    if(index < 0 || index > this.length) return false;
    if(index === 0) return !!this.unshift(val);
    if(index === this.length) return !!this.push(val);

    var newNode = new Node(val);
    var beforeNode = this.get(index-1);
    var afterNode = beforeNode.next;

    beforeNode.next = newNode, newNode.prev = beforeNode;
    newNode.next = afterNode, afterNode.prev = newNode;
    this.length++;
    return true;
}
//Elimina el nodo del indice indicado.
remove(index){
    if(index < 0 || index > this.length) return null;
    if(index === 0) return this.shift();
    if(index === this.length) return this.pop();
    var removeNode = this.get(index);
    removeNode.prev.next = removeNode.next;
    removeNode.next.prev = removeNode.prev;
    removeNode.next = null;
    removeNode.prev = null;
    this.length--;
    return removeNode;
}
```

Crear el objeto con la estructura presentada:

```
var list = new DoublyLinkedList()
```

Stacks: Una pila (stack en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»)



```
//Clase para crear un nodo.
class Node {
    constructor(value){
        this.value = value;
        this.next = null;
    }
}
//Estructura principal
class Stack {
    constructor(){
        this.first = null;
        this.last = null;
        this.size = 0;
    }
    //Agregar un nodo al stack
    push(val){
        var newNode = new Node(val);
        if(!this.first){
            this.first = newNode;
            this.last = newNode;
        } else {
            var temp = this.first;
            this.first = newNode;
            this.first.next = temp;
        }
        return ++this.size;
    }
    //Eliminar un nodo del stack.
    pop(){
        if(!this.first) return null;
        var temp = this.first;
        if(this.first === this.last){
            this.last = null;
        }
        this.first = this.first.next;
        this.size--;
        return temp.value;
    }
}
```

Queues: es una estructura de datos que sigue la Filosofía FIFO del ingles First In – First Out que en español seria “Primero en entrar primero en salir”. Esto quiere decir que el elemento que entre primero a la Cola será el primero que salga y el último que entre será el último en salir.



```
//Clase para crear un nodo.
class Node {
    constructor(value){
        this.value = value;
        this.next = null;
    }
}
//Estructura principal.
class Queue {
    constructor(){
        this.first = null;
        this.last = null;
        this.size = 0;
    }
    //Ingresar un nodo a la cola ("enqueue").
enqueue(val){
    var newNode = new Node(val);
    if(!this.first){
        this.first = newNode;
        this.last = newNode;
    } else {
        this.last.next = newNode;
        this.last = newNode;
    }
    return ++this.size;
}
    //Eliminar un nodo de la cola ("dequeue").
dequeue(){
    if(!this.first) return null;

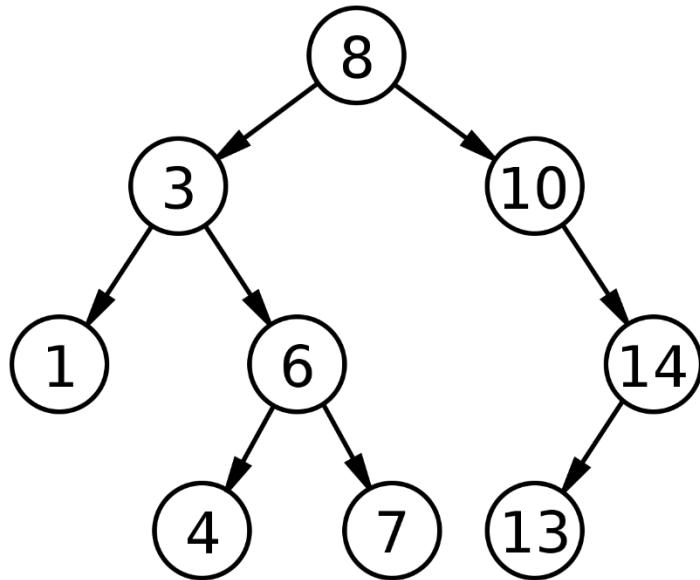
    var temp = this.first;
    if(this.first === this.last) {
        this.last = null;
    }
    this.first = this.first.next;
    this.size--;
    return temp.value;
}
}
```

Stack and Queue, time complexity:

Data Structure	Time Complexity			
	Average			
	Access	Search	Insertion	Deletion
<u>Array</u>	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
<u>Stack</u>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
<u>Queue</u>	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Binary Search Trees: Es una estructura de datos de árbol binario basada en nodos que tiene las siguientes propiedades:

- El subárbol izquierdo de un nodo contiene solo nodos con claves menores que la clave del nodo.
- El subárbol derecho de un nodo contiene solo nodos con claves mayores que la clave del nodo.
- El subárbol izquierdo y derecho también debe ser un árbol de búsqueda binaria.



```

//Clase para crear un nodo.
class Node {
    constructor(value){
        this.value = value;
        //Ahora no nos referimos al siguiente nodo o al anterior, hablamos del nodo de la izquierda o de la derecha.
        this.left = null;
        this.right = null;
    }
}
//Estructura principal
class BinarySearchTree {
    constructor(){
        this.root = null; //Solo le inicializa la raiz del arbol en nulo
    }
    //Metodo para insertar un nodo
    insert(value){
        var newNode = new Node(value);
        if(this.root === null){
            this.root = newNode;
            return this;
        }
        var current = this.root;
        while(true){
            if(value === current.value) return undefined;
            if(value < current.value){
                if(current.left === null){
                    current.left = newNode;
                    return this;
                }
                current = current.left;
            } else {
                if(current.right === null){
                    current.right = newNode;
                    return this;
                }
                current = current.right;
            }
        }
    }
    //Metodo para encontrar un nodo dentro del arbol.
    find(value){
        if(this.root === null) return false;
        var current = this.root;
        found = false;
        while(current && !found){
            if(value < current.value){
                current = current.left;
            } else if(value > current.value){
                current = current.right;
            } else {
                found = true;
            }
        }
        if(!found) return undefined;
        return current;
    }
    //Metodo que retorna verdadero o falso si el nodo buscado se encuentra dentro del arbol.
    contains(value){
        if(this.root === null) return false;
        var current = this.root;
        found = false;
        while(current && !found){
            if(value < current.value){
                current = current.left;
            } else if(value > current.value){
                current = current.right;
            } else {
                return true;
            }
        }
        return false;
    }
}

var tree = new BinarySearchTree();

```

BST, Time complexity:

Comparing Binary Search Trees to Linear Lists

Big-O Comparison			
Operation	Binary Search Tree	Array-based List	Linked List
Constructor	$O(1)$	$O(1)$	$O(1)$
Destructor	$O(N)$	$O(1)$	$O(N)$
IsFull	$O(1)$	$O(1)$	$O(1)$
IsEmpty	$O(1)$	$O(1)$	$O(1)$
RetrieveItem	$O(\log N)^*$	$O(\log N)$	$O(N)$
InsertItem	$O(\log N)^*$	$O(N)$	$O(N)$
DeleteItem	$O(\log N)^*$	$O(N)$	$O(N)$

Dentro de Binary Search Trees, tenemos dos algoritmos de búsqueda importantes: DFS (Depth First Search) y BFS (Breadth-First Search), “Estos algoritmos también pueden ser implementados en arboles normales y grafos”.

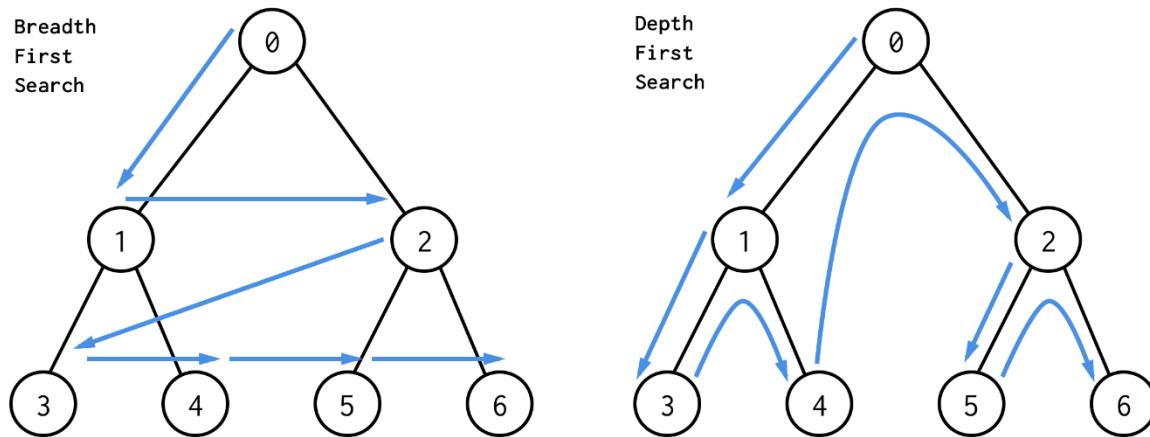
DFS: El algoritmo comienza en el nodo raíz y luego explora cada rama antes de retroceder. Se implementa mediante pilas. A menudo, mientras escribimos el código, usamos Stacks de recursión para retroceder. Al usar la recursividad, podemos aprovechar el hecho de que los subárboles izquierdo y derecho también son árboles y comparten las mismas propiedades.

Tres tipos de DFS:

- In-Order: Este método recorre los nodos del árbol en el siguiente orden: primero el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho.
- Pre-Order: Este método recorre los nodos del árbol en el siguiente orden: primero el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho.

- Post-Order: Este método recorre los nodos del árbol en el siguiente orden: primero el subárbol izquierdo, luego el subárbol derecho y finalmente el nodo raíz.

BFS: Este algoritmo también comienza en el nodo raíz y luego visita todos los nodos nivel por nivel. Eso significa que después de la raíz, atraviesa todos los hijos directos de la raíz. Después de atravesar todos los hijos directos de la raíz, se mueve a sus hijos y así sucesivamente. Para implementar BFS usamos una Queue.



```

  ⚡ ⚢ ⚣
    BFS(){
        var node = this.root,
            data = [],
            queue = [];
        queue.push(node); // Agregar el nodo raíz a la cola

        while(queue.length){
            node = queue.shift(); // Obtener el primer nodo de la cola
            data.push(node.value); // Agregar el valor del nodo a la lista de datos
            if(node.left) queue.push(node.left); // Agregar el hijo izquierdo a la cola si existe
            if(node.right) queue.push(node.right); // Agregar el hijo derecho a la cola si existe
        }
        return data; // Devolver la lista de datos recopilados
    }

```

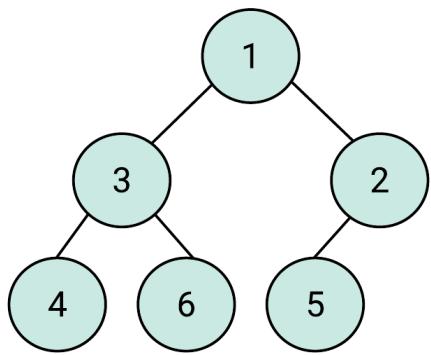
```

DFSPreOrder(){
    var data = [];
    function traverse(node){
        data.push(node.value); // Agregar el valor del nodo a la lista de datos
        if(node.left) traverse(node.left); // Recorrer el subárbol izquierdo si existe
        if(node.right) traverse(node.right); // Recorrer el subárbol derecho si existe
    }
    traverse(this.root); // Comenzar el recorrido desde la raíz
    return data; // Devolver la lista de datos recopilados
}
DFSPostOrder(){
    var data = [];
    function traverse(node){
        if(node.left) traverse(node.left);
        if(node.right) traverse(node.right);
        data.push(node.value); //Agregamos el nodo raíz al final.
    }
    traverse(this.root);
    return data;
}
DFSInOrder(){
    var data = [];
    function traverse(node){
        if(node.left) traverse(node.left);
        data.push(node.value); //Agregamos el nodo raíz luego del subárbol izquierdo.
        if(node.right) traverse(node.right);
    }
    traverse(this.root);
    return data;
}

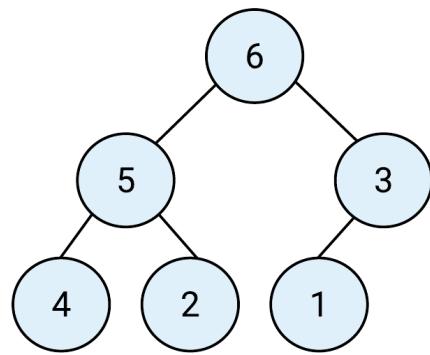
```

Binary Heaps: Es un árbol binario completo que se utiliza para almacenar datos de manera eficiente para obtener el elemento máximo o mínimo en función de su estructura.

- Min Heap
- Max Heap



Min heap



Max Heap

```

● ● ●

//Estructura Max Binary Heap.
class MaxBinaryHeap {
    constructor(){
        this.values = []; //inicializamos un array vacío.
    }
    insert(element){
        this.values.push(element); //pusheamos un elemento dentro del array values, se almacena siempre al final.
        this.bubbleUp(); //llamamos a la función bubbleUp que acomoda los elementos en su orden correspondiente de mayor a menor.
    }
    bubbleUp(){
        let idx = this.values.length - 1; //indicamos el índice del último elemento dentro del array.
        const element = this.values[idx]; //obtenemos el último elemento (elemento hijo)
        //implementamos un bucle while
        while(idx > 0){
            let parentIdx = Math.floor((idx - 1)/2); //obtengo el índice del elemento padre, mediante la fórmula matemática.
            let parent = this.values[parentIdx]; //declaro el elemento padre.
            if(element <= parent) break; //si mi elemento hijo es menor no hago nada, rompo el bucle.
            //si mi elemento hijo es mayor al padre, realizo el swap.
            this.values[parentIdx] = element;
            this.values[idx] = parent;
            idx = parentIdx;
        }
    }
    //Elimina el elemento de mayor valor "la raíz"
    extractMax(){
        if(this.values.length === 0) return undefined;
        const max = this.values[0];
        /*
        Se realiza un pop en el array de valores, se declara el elemento que se encontraba en el final como la nueva raíz, luego se lo acomoda en su lugar, con la función "sinkDown".
        */
        const end = this.values.pop();
        if(this.values.length > 0){
            this.values[0] = end;
            this.sinkDown();
        }
        return max;
    }
    sinkDown(){
        let idx = 0;
        const length = this.values.length;
        const element = this.values[0];
        while(true){
            let leftChildIdx = 2 * idx + 1; //fórmula matemática para obtener el hijo de la izquierda.
            let rightChildIdx = 2 * idx + 2;//fórmula matemática para obtener el hijo de la derecha.
            let leftChild,rightChild;
            let swap = null;

            if(leftChildIdx < length){
                leftChild = this.values[leftChildIdx];
                if(leftChild > element) {
                    swap = leftChildIdx;
                }
            }
            if(rightChildIdx < length){
                rightChild = this.values[rightChildIdx];
                if(
                    (swap === null && rightChild > element) ||
                    (swap !== null && rightChild > leftChild)
                ) {
                    swap = rightChildIdx;
                }
            }
            if(swap === null) break;
            this.values[idx] = this.values[swap];
            this.values[swap] = element;
            idx = swap;
        }
    }
}

```

Casos concretos de implementación, prioridades en una sala de emergencias:

```
● ● ●

class Node {
    constructor(val, priority){
        this.val = val;
        this.priority = priority;
    }
}

class PriorityQueue {
    constructor(){
        this.values = [];
    }
    enqueue(val, priority){
        let newNode = new Node(val, priority);
        this.values.push(newNode);
        this.bubbleUp();
    }
    bubbleUp(){
        let idx = this.values.length - 1;
        const element = this.values[idx];
        while(idx > 0){
            let parentIdx = Math.floor((idx - 1)/2);
            let parent = this.values[parentIdx];
            if(element.priority >= parent.priority) break;
            this.values[parentIdx] = element;
            this.values[idx] = parent;
            idx = parentIdx;
        }
    }
    dequeue(){
        const min = this.values[0];
        const end = this.values.pop();
        if(this.values.length > 0){
            this.values[0] = end;
            this.sinkDown();
        }
        return min;
    }
    sinkDown(){
        let idx = 0;
        const length = this.values.length;
        const element = this.values[0];
        while(true){
            let leftChildIdx = 2 * idx + 1;
            let rightChildIdx = 2 * idx + 2;
            let leftChild,rightChild;
            let swap = null;

            if(leftChildIdx < length){
                leftChild = this.values[leftChildIdx];
                if(leftChild.priority < element.priority) {
                    swap = leftChildIdx;
                }
            }
            if(rightChildIdx < length){
                rightChild = this.values[rightChildIdx];
                if(
                    (swap === null && rightChild.priority < element.priority) ||
                    (swap !== null && rightChild.priority < leftChild.priority)
                ) {
                    swap = rightChildIdx;
                }
            }
            if(swap === null) break;
            this.values[idx] = this.values[swap];
            this.values[swap] = element;
            idx = swap;
        }
    }
}

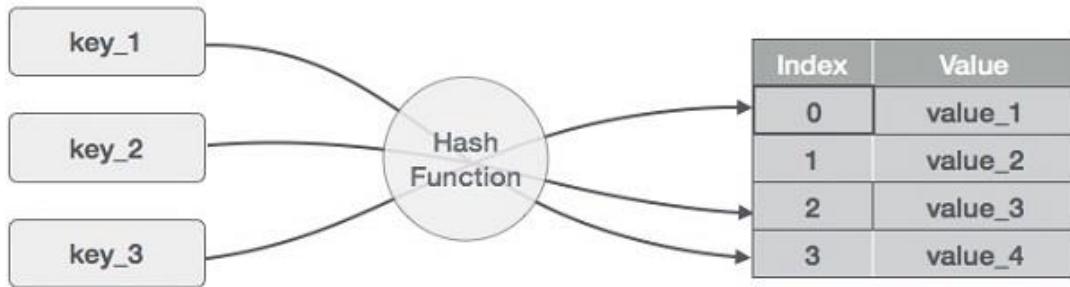
let ER = new PriorityQueue();
ER.enqueue("common cold",5)
ER.enqueue("gunshot wound", 1)
ER.enqueue("high fever",4)
ER.enqueue("broken arm",2)
ER.enqueue("glass in foot",3)
```

Time complexity de Binary Heaps: Heaps

Heaps	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)

Hash Table: Es una estructura de datos que almacena datos de manera asociativa. En una tabla hash, los datos se almacenan en un formato de matriz, donde cada valor de datos tiene su propio valor de índice único. El acceso a los datos se vuelve muy rápido si conocemos el índice de los datos deseados.

Concepto importante: Hashing es una técnica para convertir un rango de valores clave en un rango de índices de una matriz.



```

class HashTable {
    /*El método constructor define la tabla hash y su tamaño, que por defecto es 53.
    Cada elemento de la tabla es un array que puede contener varios pares clave-valor. */
    constructor(size=53){
        this.keyMap = new Array(size);
    }
    /*
    El método _hash recibe una clave y la convierte en un índice de la tabla hash utilizando un algoritmo de
    hashing.
    En este caso, se utiliza una constante WEIRD_PRIME igual a 31, que se multiplica por el valor ASCII de cada
    carácter
    de la clave y se suma a un valor acumulado. El resultado se reduce módulo el tamaño de la tabla hash.
    */
    _hash(key) {
        let total = 0;
        let WEIRD_PRIME = 31;
        for (let i = 0; i < Math.min(key.length, 100); i++) {
            let char = key[i];
            let value = char.charCodeAt(0) - 96
            total = (total * WEIRD_PRIME + value) % this.keyMap.length;
        }
        return total;
    }
    /*
    El método set recibe una clave y un valor, calcula su índice en la tabla hash mediante el método _hash,
    y agrega un nuevo par clave-valor al array correspondiente. Si el array no existe todavía, se crea uno nuevo.
    */
    set(key,value){
        let index = this._hash(key);
        if(!this.keyMap[index]){
            this.keyMap[index] = [];
        }
        this.keyMap[index].push([key, value]);
    }
    /*
    El método get recibe una clave, calcula su índice en la tabla hash mediante el método _hash,
    y busca el valor correspondiente dentro del array de pares clave-valor. Si encuentra la clave,
    devuelve el valor correspondiente. Si no la encuentra, devuelve undefined.
    */
    get(key){
        let index = this._hash(key);
        if(this.keyMap[index]){
            for(let i = 0; i < this.keyMap[index].length; i++){
                if(this.keyMap[index][i][0] === key) {
                    return this.keyMap[index][i][1];
                }
            }
        }
        return undefined;
    }
    /*
    El método keys recorre todos los elementos de la tabla hash y agrega las claves únicas a un array keysArr.
    Devuelve este array al finalizar.
    */
    keys(){
        let keysArr = [];
        for(let i = 0; i < this.keyMap.length; i++){
            if(this.keyMap[i]){
                for(let j = 0; j < this.keyMap[i].length; j++){
                    if(!keysArr.includes(this.keyMap[i][j][0])){
                        keysArr.push(this.keyMap[i][j][0]);
                    }
                }
            }
        }
        return keysArr;
    }
    /*
    El método values recorre todos los elementos de la tabla hash y agrega los valores únicos a un array
    valuesArr.
    Devuelve este array al finalizar.
    */
    values(){
        let valuesArr = [];
        for(let i = 0; i < this.keyMap.length; i++){
            if(this.keyMap[i]){
                for(let j = 0; j < this.keyMap[i].length; j++){
                    if(!valuesArr.includes(this.keyMap[i][j][1])){
                        valuesArr.push(this.keyMap[i][j][1]);
                    }
                }
            }
        }
        return valuesArr;
    }
}

```

Time complexity de Hash Table:

Hash table		
Type	Unordered associative array	
Invented	1953	
Time complexity in big O notation		
Algorithm	Average	Worst case
Space	$O(n)$ ^[1]	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Graph: es una estructura de datos no lineal que consta de vértices y aristas. Los vértices a veces también se denominan nodos y los bordes son líneas o arcos que conectan dos nodos en el gráfico. Más formalmente, un gráfico se compone de un conjunto de vértices (V) y un conjunto de aristas (E). El gráfico se denota por $G(E, V)$.

```

/*Este código define una clase Graph que se utiliza para representar un grafo no dirigido mediante listas de
adyacencia.*/
class Graph{
/*
    El método constructor inicializa un objeto adjacencyList como un diccionario vacío,
    que se utilizará para almacenar la información de los vértices y sus aristas.
*/
constructor(){
    this.adjacencyList = {};
}
/*
    El método addVertex(vertex) se utiliza para agregar un vértice al grafo.
    Si el vértice aún no existe en la lista de adyacencia, se crea una nueva entrada en el diccionario
    con el nombre del vértice como clave y una lista vacía como valor.
*/
addVertex(vertex){
    if(!this.adjacencyList[vertex]) this.adjacencyList[vertex] = [];
}
/*
    El método addEdge(v1, v2) se utiliza para agregar una arista entre dos vértices en el grafo.
    Primero, se busca en el diccionario la entrada correspondiente a v1 y se agrega v2 a la lista de
    adyacencia.
    Luego, se busca en el diccionario la entrada correspondiente a v2 y se agrega v1 a su lista de
    adyacencia.
*/
addEdge(v1,v2){
    this.adjacencyList[v1].push(v2);
    this.adjacencyList[v2].push(v1);
}
/*
    El método removeEdge(vertex1, vertex2) se utiliza para eliminar una arista entre dos vértices en el
    grafo. Para hacerlo,
    busca en la lista de adyacencia de vertex1 y elimina vertex2 de la lista, y luego hace lo mismo
    en la lista de adyacencia de vertex2.
*/
removeEdge(vertex1,vertex2){
    this.adjacencyList[vertex1] = this.adjacencyList[vertex1].filter(
        v => v !== vertex2
    );
    this.adjacencyList[vertex2] = this.adjacencyList[vertex2].filter(
        v => v !== vertex1
    );
}
/*
    El método removeVertex(vertex) se utiliza para eliminar un vértice y todas sus aristas del grafo.
    Primero, mientras la lista de adyacencia del vértice aún tenga elementos, se extrae el último vértice
    adyacente
    y se llama al método removeEdge() para eliminar la arista entre el vértice y su adyacente.
    Luego, se elimina la entrada correspondiente al vértice del diccionario adjacencyList.
*/
removeVertex(vertex){
    while(this.adjacencyList[vertex].length){
        const adjacentVertex = this.adjacencyList[vertex].pop();
        this.removeEdge(vertex, adjacentVertex);
    }
    delete this.adjacencyList[vertex]
}
}

```

Si utilizamos esta estructura para crear una red social:

`addVertex` = agregar una cuenta.

`addEdge` = conectar dos perfiles.

`removeEdge` = cortar relación entre esos dos perfiles.

`removeVertex`= eliminar la cuenta de la red social.

Existen graph unidireccionales y bidireccionales.

Para recorrer un graph utilizamos los algoritmos DFS y BFS:

```
depthFirstRecursive(start){  
    // Inicializar un arreglo vacío para guardar el orden de visita de los nodos  
    const result = [];  
    // Inicializar un objeto para llevar registro de los nodos visitados  
    const visited = {};  
    // Copiar la lista de adyacencia del grafo a una variable  
    const adjacencyList = this.adjacencyList;  
  
    // Definir una función anónima recursiva para implementar el algoritmo DFS  
    (function dfs(vertex){  
        // Si el vértice actual es nulo, retornar nulo  
        if(!vertex) return null;  
        // Marcar el vértice actual como visitado  
        visited[vertex] = true;  
        // Agregar el vértice actual al arreglo de resultados  
        result.push(vertex);  
        // Recorrer la lista de adyacencia del vértice actual  
        adjacencyList[vertex].forEach(neighbor => {  
            // Si el vecino actual no ha sido visitado, llamar recursivamente a la función dfs  
            if(!visited[neighbor]){  
                return dfs(neighbor)  
            }  
        });  
    })(start); // Invocar la función dfs con el vértice de inicio  
  
    // Retornar el arreglo de resultados con el orden de visita de los nodos  
    return result;  
}
```

```
breadthFirst(start){  
    // Inicializar una cola con el vértice de inicio  
    const queue = [start];  
    // Inicializar un arreglo vacío para guardar el orden de visita de los nodos  
    const result = [];  
    // Inicializar un objeto para llevar registro de los nodos visitados  
    const visited = {};  
    // Marcar el vértice de inicio como visitado  
    visited[start] = true;  
    let currentVertex;  
  
    // Mientras la cola no esté vacía  
    while(queue.length){  
        // Sacar el primer elemento de la cola y asignarlo a la variable currentVertex  
        currentVertex = queue.shift();  
        // Agregar el vértice actual al arreglo de resultados  
        result.push(currentVertex);  
  
        // Recorrer la lista de adyacencia del vértice actual  
        this.adjacencyList[currentVertex].forEach(neighor => {  
            // Si el vecino actual no ha sido visitado, marcarlo como visitado y agregarlo a la cola  
            if(!visited[neighor]){  
                visited[neighor] = true;  
                queue.push(neighor);  
            }  
        });  
    }  
    // Retornar el arreglo de resultados con el orden de visita de los nodos  
    return result;  
}
```

Dijkstra algoritmo: El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.

```

/*
La función Dijkstratoma dos argumentos, start y finish, que representan los nodos inicial
y final respectivamente. Inicializa una cola de prioridad para almacenar los nodos, un objeto
para almacenar las distancias, otro objeto para almacenar el nodo anterior en la ruta y
dos variables path y smallest.
*/
Dijkstra(start, finish){
    const nodes = new PriorityQueue();
    const distances = {};
    const previous = {};
    let path = []
    let smallest;
    /*
        En el siguiente bucle, las distancias y los nodos anteriores se inicializan para cada vértice del
        gráfico.
        Si el vértice es el nodo de inicio, la distancia se establece en cero y se agrega a la cola de
        prioridad
        con una prioridad de cero. De lo contrario, la distancia se establece en infinito y se agrega a la cola
        de prioridad con una prioridad de infinito. El nodo anterior se establece en nulo para cada vértice.
    */
    for(let vertex in this.adjacencyList){
        if(vertex === start){
            distances[vertex] = 0;
            nodes.enqueue(vertex, 0);
        } else {
            distances[vertex] = Infinity;
            nodes.enqueue(vertex, Infinity);
        }
        previous[vertex] = null;
    }
    /*
        El siguiente bucle se ejecuta hasta que la cola de prioridad está vacía. Saca de la cola el nodo más
        pequeño
        de la cola de prioridad y comprueba si es el nodo de destino. Si es así, la ruta se construye
        retrocediendo
        a través de los nodos anteriores. Si no es el nodo de destino, el algoritmo explora sus vecinos
        y actualiza sus distancias y nodos anteriores si encuentra una ruta más corta.
        Los nodos vecinos se agregan a la cola de prioridad con sus distancias actualizadas.
    */
    while(nodes.values.length){
        smallest = nodes.dequeue().val;
        if(smallest === finish){
            while(previous[smallest]){
                path.push(smallest);
                smallest = previous[smallest];
            }
            break;
        }
        if(smallest || distances[smallest] !== Infinity){
            for(let neighbor in this.adjacencyList[smallest]){
                let nextNode = this.adjacencyList[smallest][neighbor];

                let candidate = distances[smallest] + nextNode.weight;
                let nextNeighbor = nextNode.node;
                if(candidate < distances[nextNeighbor]){
                    distances[nextNeighbor] = candidate;
                    previous[nextNeighbor] = smallest;
                    nodes.enqueue(nextNeighbor, candidate);
                }
            }
        }
    }
    return path.concat(smallest).reverse();
}

```

Dynamic Programming

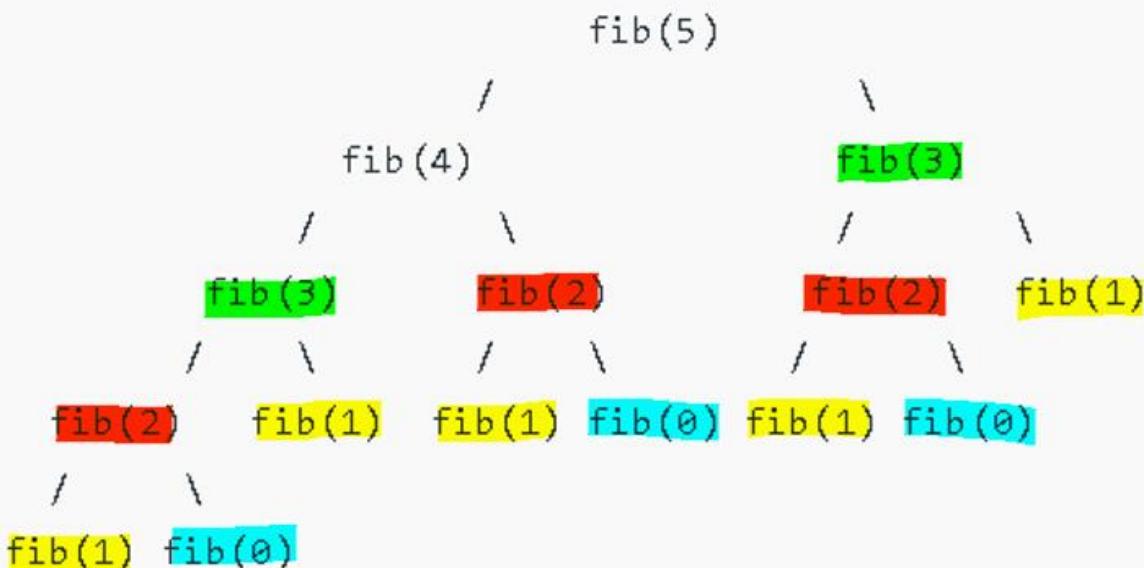
La programación dinámica es principalmente una optimización sobre recursividad simple. Donde quiera que veamos una solución recursiva que tiene llamadas repetidas para las mismas entradas, podemos optimizarla usando Programación Dinámica. La idea es simplemente almacenar los resultados de los subproblemas, para que no tengamos que volver a calcularlos cuando sea necesario más adelante. Esta sencilla optimización reduce las complejidades del tiempo de exponencial a polinomial.

overlapping subproblems: Se dice que un problema tiene subproblemas superpuestos si se

puede dividir en subproblemas que se reutilizan varias veces.

Ejemplo: Calcular el valor de un numero de Fibonacci dado su posición.

Al intentar calcular por ejemplo la posición número 5 dentro de la secuencia de Fibonacci y resolver el problema utilizando recursividad, varias veces debemos utilizar el valor de $\text{fib}(0), \text{fib}(1), \text{fib}(2), \text{fib}(3)$. En vez de recalcularlo podemos almacenarlos de alguna manera para simplificar el tiempo de complejidad.



optimal substructure: Se dice que un problema tiene una subestructura óptima si se puede construir una solución óptima a partir de soluciones óptimas de sus subproblemas.

Fib(5) depende de Fib(0), Fib(1), Fib(2), Fib(3), Fib(4).

Ahora supongamos el subproblema de encontrar Fib(4), este también depende de Fib(0), Fib(1), Fib(2), Fib(3). Es decir que el problema de los números Fibonacci cumple con una subestructura optimizada.

Memoization: En informática, la memoización se utiliza para acelerar los programas informáticos al eliminar el cálculo repetitivo de los resultados y al evitar llamadas repetidas a funciones que procesan la misma entrada.

```
//Solución para la función de fibonacci con dynamic programming
fib(n,memo[]){
    if(memo[n]!==undefined) return memo[n]
    if(n<=2) return 1
    var res = fib(n-1 , memo) + fib(n-2, memo);
    memo[n] = res;
    return res;
}
```