# Estructuras de datos

## ¿Qué es buen código?

Es el código que tiene en cuenta:

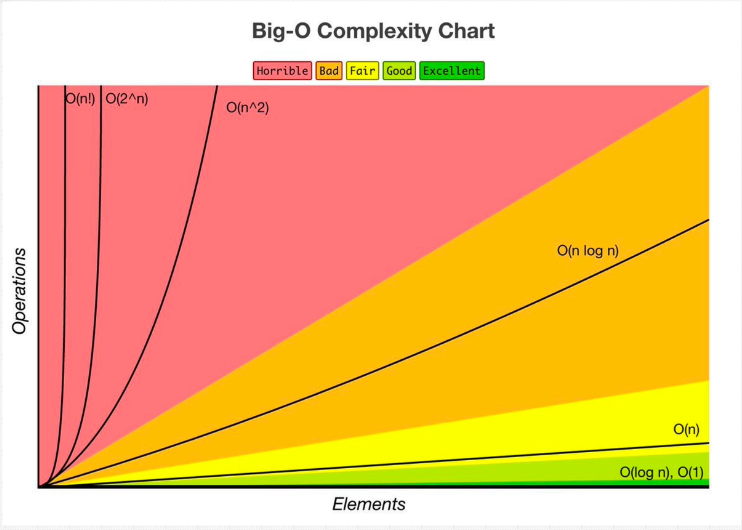
1. Legibilidad.
2. Veloz.
3. Memoria.

## ¿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

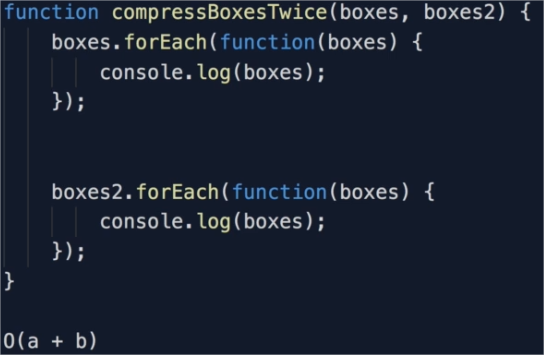
Siempre se calcula en base al peor caso.

#### 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.



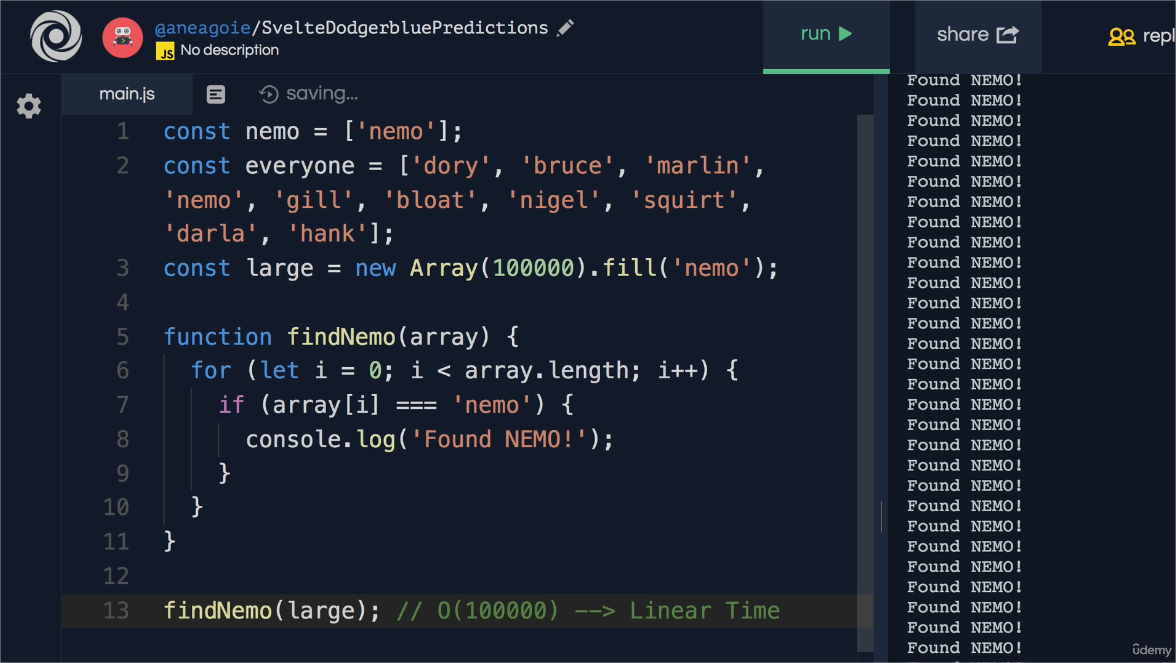
#### Regla 4: Elimina los no dominantes

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)

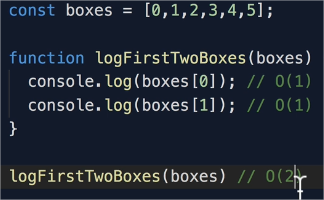
### O(n)

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**.

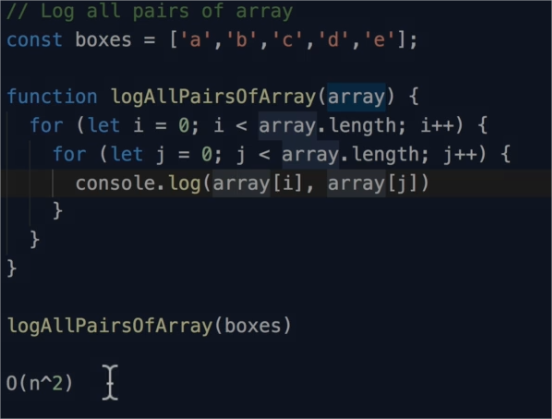
### O(1)

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.

### O(log n)

Es la notación logarítmica. Es cuando, en lugar de recorrer cada elemento de una lista (O(n)), se “divide y conquista”.

## 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++) {

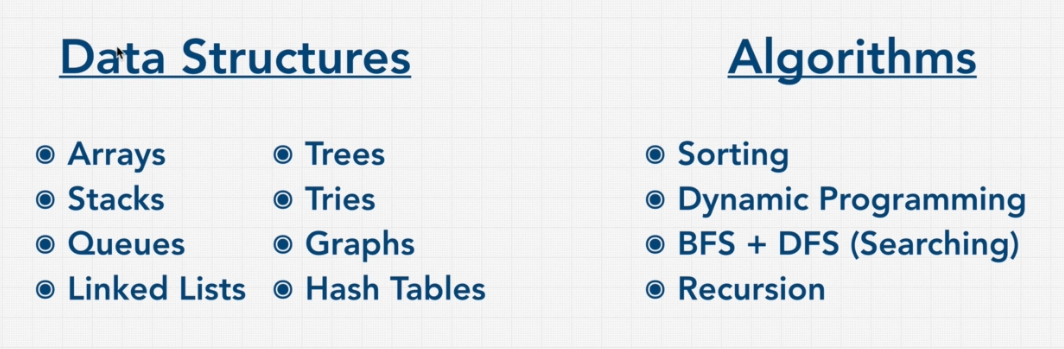
hiArray[i] = 'hi'; //O(n)

}

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)

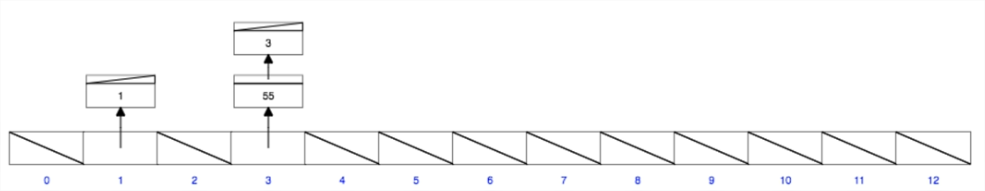
Search: O(1)

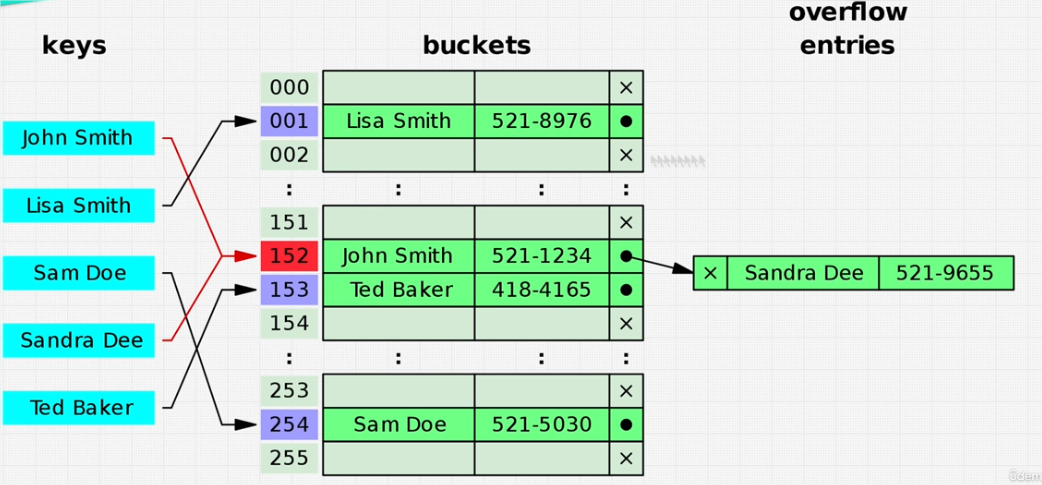
#### Colisión Hash

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.

### Linked lists

Como dice el nombre, es una lista enlazada.

La lista contiene un set de nodos, estos nodos tienen dos partes:

* El valor del dato
* El puntero del siguiente nodo

El primer nodo se llama “head” y el último nodo se llama “tail”.

Las listas enlazadas son terminadas en null, quiere decir que el puntero del nodo tail apuntará a null.

Pueden ser listas ordenadas o desordenadas.

Prepend (Agregar al principio): O(1)

Append: O(1)

Lookup: O(n)

Insert: O(n)

Delete: O(n)

#### Puntero

Un puntero es una secuencia de caracteres que apunta a una dirección en memoria.

#### Doubly Linked Lists

Las doubly linked lists son linked list donde los nodos tienen una propiedad más que es un puntero que apunta al nodo anterior.



Esto nos permite recorrer la lista hacía atrás.

#### Review

Pros:

* Inserción y eliminación rápida
* Ordenadas
* Tamaño flexible

Contras:

* Búsqueda lenta
* Más memoria

### Stacks

Es Puedes pensar en los elementos de los stack como una “pila de platos”, ya que no tienes acceso a los platos de abajo sin antes sacar al que está arriba de todo.

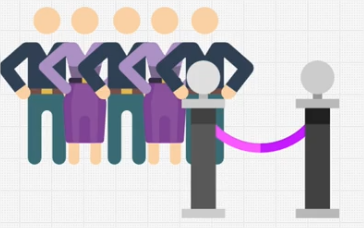
Esto es llamado LIFO (Last In First Out), que significa que el último en entrar es el primero en salir.

Los métodos que utiliza y sus Big O son los siguientes:

* Lookup: O(n)
* Push (Agregar un elemento): O(1)
* Pop (Sacar el último elemento): O(1)
* Peek (Obtener el último elemento): O(1)

### Queues

Se pueden pensar en los elementos de las queues como “colas de gente”.

Esto es llamado FIFO (First In First Out), que significa que el primero en entrar es el primero en salir.

Los métodos que utiliza y sus Big O son los siguientes:

* Lookup: O(n) - Dequeue (Quitar el siguiente elemento de la lista): O(1)
* Enqueue (Agregar un elemento): O(1) - Peek: O(1)

### Stacks y Queues

Ambas son estructuras de datos linear. Eso significa que nos permite ir a través de los elementos de la estructura de datos secuencialmente uno por uno.

#### Review

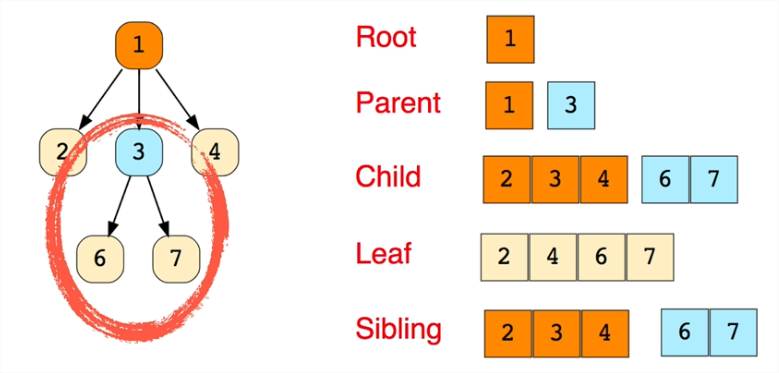
Pros:

* Operaciones rápidas
* Funciones peek rápidas
* Ordenadas

Contras:

* Búsqueda lenta

### Trees

Son una estructura de datos que tiene una estructura jerarquica. Esto es diferente a las linked lists o arrays que tienen una estructura linear  


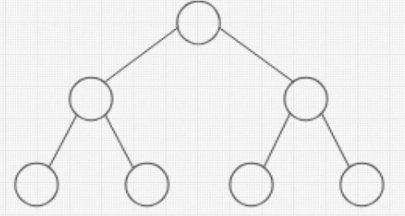
#### Binary Trees

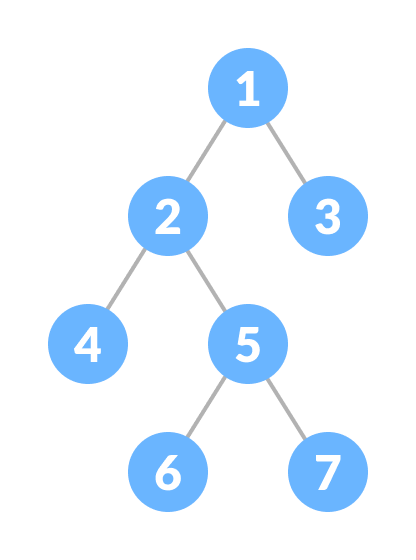
Un árbol binario, cada nodo puede tener 0, 1 o 2 nodos hijos y solo 1 padre.

Por ejemplo:

Este es un árbol binario Este NO es un árbol binario

#### Terminología

* Perfecto binary tree (Árbol binario perfecto): Es el árbol donde cada nodo interno tiene exactamente dos nodos hijos y los leaf nodes están al mismo nivel
* Full binary tree (Árbol binario lleno): Es cuando un árbol binario tiene o 0 o 2 hijos, pero NUNCA 1. También es llamado proper binary tree.

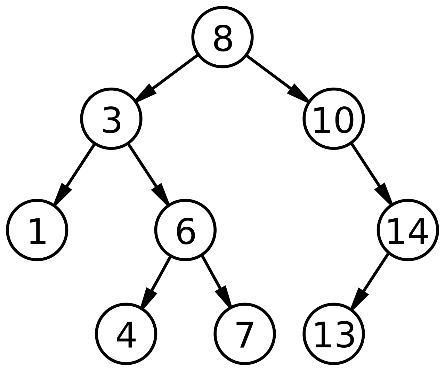


* Leaf node (Nodo hoja): Son los nodos que no tienen hijos.

#### Binary Search Tree

Es una estructura de datos donde los nodos hijos del lado derecho son mayores al nodo padre y los izquierdos son menores.

Por ejemplo

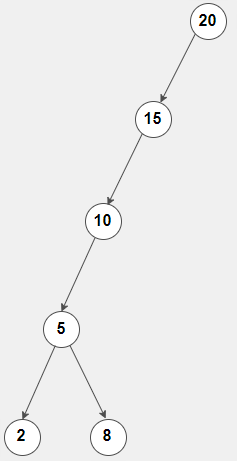


Los métodos que utiliza y sus Big O son los siguientes:

* Lookup: O(log N)
* Insert: O(log N)
* Delete: O(log N)

Otra regla que tiene, es que cada nodo solo puede tener dos hijos.

#### Unbalanced Binary Search Tree

Es cuando todos los nodos de un árbol se van para un solo lado del arbol

Esto hace que aumente los tiempos de sus métodos:

* Lookup: O(n)
* Insert: O(n)

#### Delete: O(n) Review

Pros:

* Son mejores que O(n)
* Son ordenadas
* Tienen tamaño flexible

Contras:

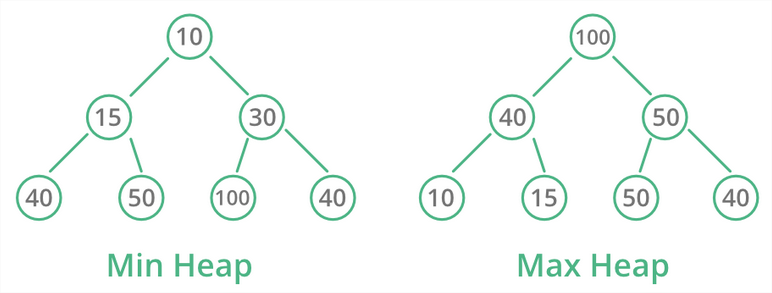
* No tienen operaciones O(1)

#### AVL Trees + Red Black Trees

Son arboles binarios que se balancean automáticamente.

#### Binary Heap

En un Binary Heap, cada hijo pertenece a un nodo padre que tiene una más alta prioridad o valor si es Max Heap y menor prioridad o valor si es Min Heap.



Sus métodos y tiempos son:

* Lookup: O(n) - Delete: O(log n)
* Insert: O(log n)

Lo bueno de binary heap es que ocupan la menor cantidad de espacio en memoria posible debido a que se insertan de izquierda a derecha.

Gracias a esto último, no existe el concepto de un binary heap desbalanceado.

#### Review

Pros:

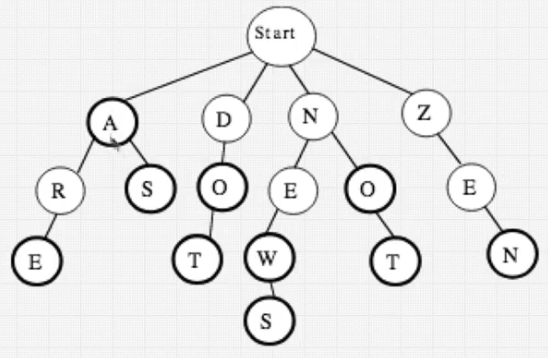
* Mejor que O(n)
* Prioridad
* Tamaño flexible
* Inserts rápidos

Contras:

* Búsqueda lenta

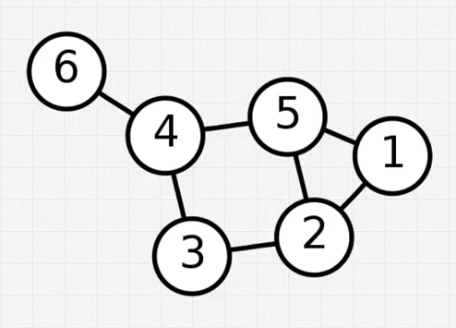
#### Trie

Es un árbol especializado en la búsqueda y utilizado normalmente con texto.

Sirven para cosas como ofrecer sugerencias o ser un diccionario a medida que se va especificando el texto.  


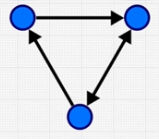
### Graphs (Grafos)

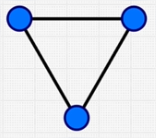
Es una estructura de datos donde cada valor es un nodo/vértice, y están conectado por aristas (edge).



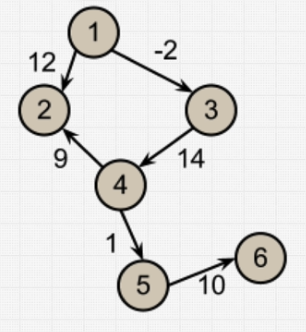
Son bastante útiles para modelar relaciones del mundo real.

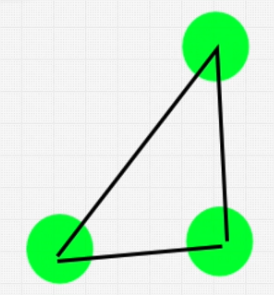
Hay varios tipos de grafos:

Direccionados No direccionados



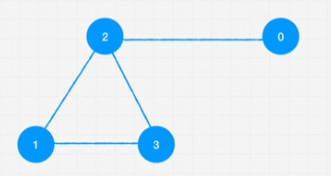
Sin peso Con peso



Ciclicos Aciclicos

#### Formas de representar un grafo con arrays

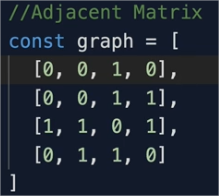
Grafo a representar:



Formas:







#### Review

Pros:

* Relaciones entre nodos

Contras:

* Escalarlo es difícil

# Algoritmos

Los algoritmos son simplemente pasos en un proceso que tomamos para ejecutar una determinada acción con los ordenadores.

Los programas son algoritmos + estructuras de datos.

## Recursión

Una función recursiva, es una función que se hace referencia a sí misma.

Reglas:

1. Identificar el caso base (el caso donde la función termina).
2. Identificar el caso recursivo.
3. En el caso recursivo, devolver con un return la función recursiva, así devuelve el resultado.

Todo lo que pueda ser implementado recursivamente, puede ser implementado iterativamente.

### Review

Pros:

* Limpio
* Legible

Contra:

* Stack grande (capaz de dar stack overflow si no está bien calculado).

### Cuando usar recursión

Cuando hay problemas complicados, como recorrer o buscar en un árbol o grafo, la recursión es bastante útil.

Cada vez que estés usando un árbol o convirtiendo algo a un árbol, considera la recursión:

1. Dividelo en un numero de subproblemas que son instancias más pequeñas del mismo problema.
2. Cada instancia del subproblema es idéntico en naturaleza.
3. La solución de cada subproblema puede ser combinada para resolver el problema en la mano.

## Sorting (Ordenado)

Hay varios algoritmos de ordenados, se explicarán algunos de ellos.

### Bubble sort

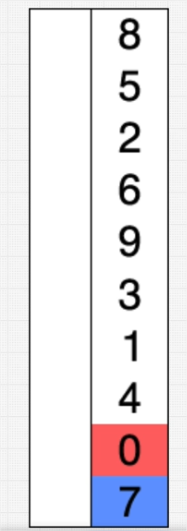
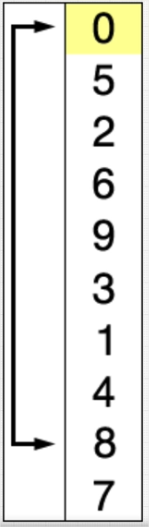
Implica comparar dos números de la lista y el mayor va escalando en la lista, de esta manera con cada iteración se va ordenando.

Por esto se llama “Bubble sort” porque el número es como una “burbuja” que va subiendo.



### Selection sort

Implica escanear una lista de elementos para buscar el menor elemento y cambiarlo por la primera posición. Y así con el resto de elementos

### Insertion sort

Es útil cuando estás seguro de que la lista está poco desordenada u ordenada, y en el best-case scenario, se podrá obtener un O(n) a la hora de ordenar.

Consiste en recorrer una lista y, en cada elemento, verificar si el anterior es menor e ir intercambiando posiciones con el anterior. Si el anterior del elemento cambiado es también menor, cambia posiciones y así.



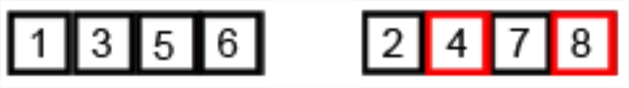


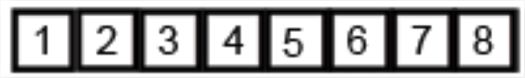
En resumen, consiste en ordenar los números a la izquierda del número que se está recorriendo. Eso es lo que lo vuelve rápido en el mejor escenario.

### Merge Sort

Consiste separar y dividir. La lista de elementos se separa hasta que sean varias “listas” de un elemento. Luego, se van ordenando de dos en dos, de cuatro en cuatro y así.

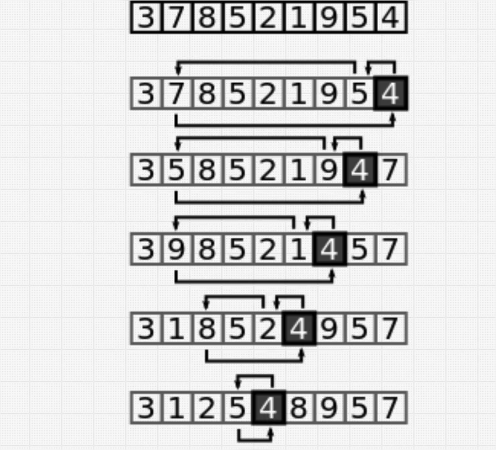




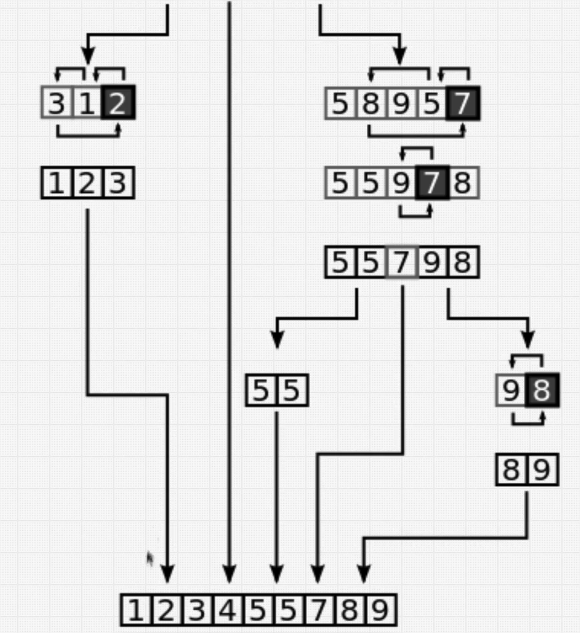


### Quick Sort

Consiste en coger un número de “pivote” y, los elementos que sean menores a este número van a la izquierda y los que sean mayores van a la derecha.



Una vez separados los menores y mayores usando el pivote, se separan los menores y mayores pero esta vez en listas, se escoge un pivote y se vuelve a realizar la secuencia.

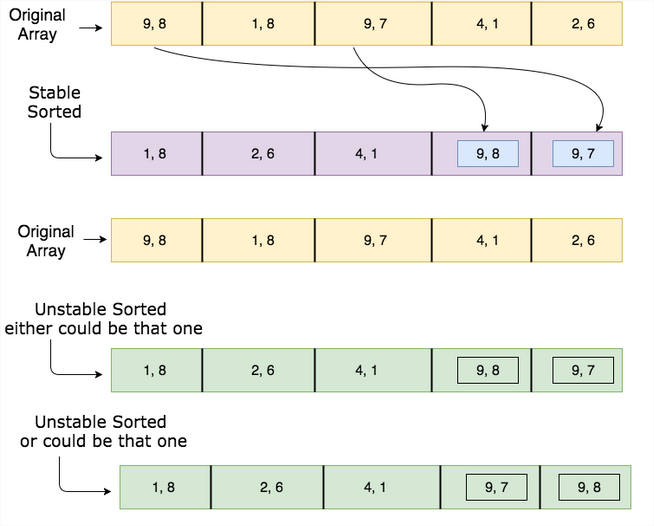


### Algoritmos estables e inestables

Los algoritmos estables son los que, al ordenar, tienen en cuenta el objeto al completo.

Por ejemplo, al ordenar una lista de 5 elementos donde cada elemento es una palabra de 5 letras, se ordenaría correctamente incluso cuando haya palabras parecidas como “straw” o “spork”.

Un algoritmo inestable no puede prometer que esas palabras estarán ordenadas, ya que solo se fija, en este caso, por la primera letra, que la toma como “llave”.



## Searching (Búsqueda)

### Linear search

También llamada búsqueda secuencial, es el método de búsqueda que consiste en buscar secuencialmente en cada elemento de la lista.

### Binary search

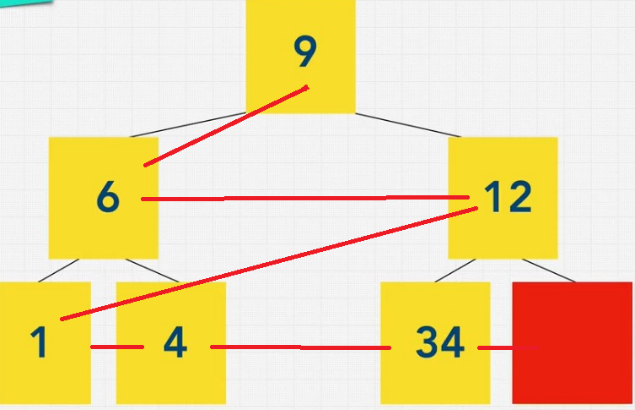
Es el método de búsqueda que consiste en empezar la búsqueda desde la mitad de la lista.

Si el elemento seleccionado es menor, se obtiene la mitad de la derecha, si es mayor, se obtiene la mitad de la izquierda.

Una vez obtenida la mitad que contiene el número, se repite el proceso.

### Breadth First Search/Traversal

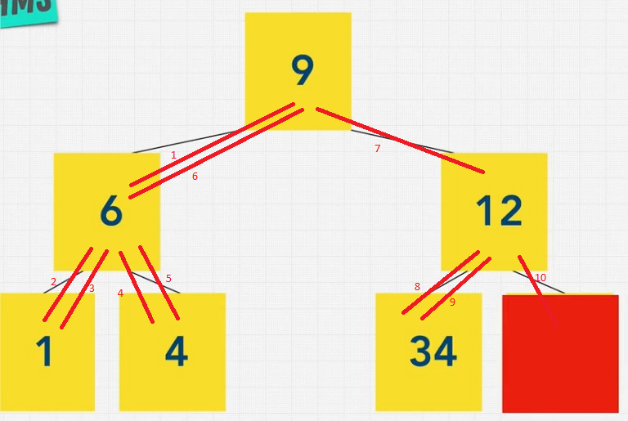
Es un método de búsqueda que consiste en recorrer cada nivel del árbol binario/grafo de manera horizontal hasta encontrar el elemento buscado.



Usa memoria adicional porque es necesario guardar el nodo tail de todos los nodos del nivel que se está recorriendo.

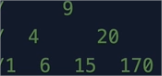
### Depth First Search/Traversal

Consiste en visitar cada rama hasta encontrarse con el nodo leaf.



Usa menos memoria que el BFS porque no es necesario guardar los punteros de los nodos hijos.

DFS tiene tres formas de recorrer un arbol:



#### InOrder

El recorrido empieza desde el nodo más izquierdo posible, luego va al nodo padre y luego a la derecha. En este caso sería 1, 4, 6, 9, 15, 20, 170.

#### PreOrder

El recorrido empieza desde el padre, luego izquierda y luego derecha. En este caso sería 9, 4, 1, 6, 20, 15, 170

#### PostOrder

El recorrido empieza desde el nodo más izquierdo, luego al nodo de la derecha y luego al nodo padre. En este caso sería 1, 6, 4, 15, 170, 20, 9.

### BFS vs DFS

BFS pros:

* Bueno para buscar el camino más corto
* Nodos más cercanos

BFS contras:

* Más memoria

DFS pros:

* Menos memoria
* Es bueno para verificar si existe un camino

DFS contras:

* Puede volverse más lento

### Dijkstra y Bellman-Ford

Son algoritmos de grafo encargado de buscar el camino más corto cuando son grafos con peso.

Bellman es mejor que Dijkstra para buscar el camino más corto ya que puede lidiar con grafos con pesos negativos.

Sin embargo, Bellman tiene una complejidad de tiempo de O(n^2). En cambio, Dijkstra es más rápido que eso.

Entonces Bellman se utiliza para grafos con peso negativo, si no hay grafos con peso negativo, entonces se utiliza Dijkstra.

## Programación dinámica

Es una técnica de optimización, si hay algo que se puede cachear, entonces se puede utilizar la programación dinámica.

### Caching

Es una manera de guardar valores para utilizarlo más tarde.