Apuntes Ciencias de la Computación

Estructura de Datos y Algoritmo

Temario

- Órdenes de complejidad computacional
- Estructuras básicas: Listas, Pilas, Colas
- Otras estructuras: Hash tables, grafos, árboles, heaps
- Algoritmos de ordenamiento: BubbleSort, InsertSort, HeapSort, MergeSort, QuickSort
- Clases comunes de algoritmos: Divide and conquer, greedy algorithms, dynamic programming
- Búsqueda de soluciones: DFS, BFS, Backtracking

Orden de complejidad

Los órdenes de complejidad hacen referencia al costo computacional del algoritmo, están dispuestos en 3 medidas O, Ω, θ.

Estas medidas se refieren a f(n) donde n representa el tamaño de algún elemento del problema a resolver mediante el algoritmo.

- Superior asintótica (O): Se utiliza para encontrar la función del peor caso.
- Cota inferior asintótica (Ω): Se utiliza para calcular el mejor caso para los algoritmos
- Cota ajustada asintótica (θ): Es la función promedio de un algoritmo.

Estructuras fundamentales

LISTAS

Dato + Referencia

- **Ligada o enlazada**: contiene el dato y la información del sucesor, pero no del antecesor, el ultimo miembro de la lista apunta a null.
  - No tienen un largo máximo, seguir agregando mientras tenga espacio en memoria

- **Indexada**: Arreglos de toda la vida

- **Doblemente Ligada**: Contiene el dato y la información del sucesor y del antecesor, el ultimo apunta a null y el primero(root) apunta también.
  - Circular: Es una lista doblemente enlazada, pero el root apunta al ultimo y el ultimo apunta al root.
  - Indexada (Array): Es un arreglo porque cada miembro tiene un índice.

Diferencias →

- ligadas hay que recorrer por todos para llegar,
- en las indexadas se refiere nomas a[2] y se acaba, se accede al elemento ,
- Si se saca un elemento se tienen que arreglar las referencias

- **Hash (lista asociativa):**
- **Pila -> LIFO (Last in First Out)**

  Se agregan a uno posterior a la referencia del último, (push en el stack , se actualiza el puntero)
  Puntero para el ultimo (con ese se juega), actualizar punteros.
  Pop -> se saca y se retorna este de la lista

- **Cola -> FIFO (Fist in First Out)**

Operaciones

- Lista:
  - prepend(X) -> lo contrario de append, agrega al incio
  - append (X) -> ingresar al final
  - remove(X) -> eliminar
  - head() -> cabeza
  - tail() -> cola
  - get (i) -> recorrer hasta obtener este elemento y retrnar
- Pila:
  - push (X)
  - pop()
  - peek () -> pop pero no quita el elemento, mirando el tope de la pila
- Cola:
  - enqueue (X) -> ingresar algo a la cola
  - equeue () -> quitar algo de la cola

TIPOS ESPECIALES

Circular → lista que el ultimo apunta al primero como el siguiente

Cola priorizada → prioridad según un adjetivo de esta cola un peso

Cola deque → doble ended queue, la cola tiene dos limites, una por cada lado .

// funciones recursivas conviene usar una pila. cuando se va resolviendo se van apilando y al ir resolviendo la recursión se van des apilando

// colas → en comunicaciones, peticiones de router , llegando peticiones, etc.

Tablas de Hash

Generalmente están en el orde O de 1, son excelentes para buscar cosas

Mapas o diccionarios asociativos, son arreglos en los que se reemplaza el índice

Son muy buenas para buscar cosas, valor a llave

h(v) -> k // Par país -> capital

h es la función de hash

Duración -> O de 1 ,

Colisiones cuando un mismo valor apunta a la misma llave, listas ligadas, etc... , búsquedas dentro de la llave, **encadenamiento(chaining)**, una cadena de valores asociadas a una llave.

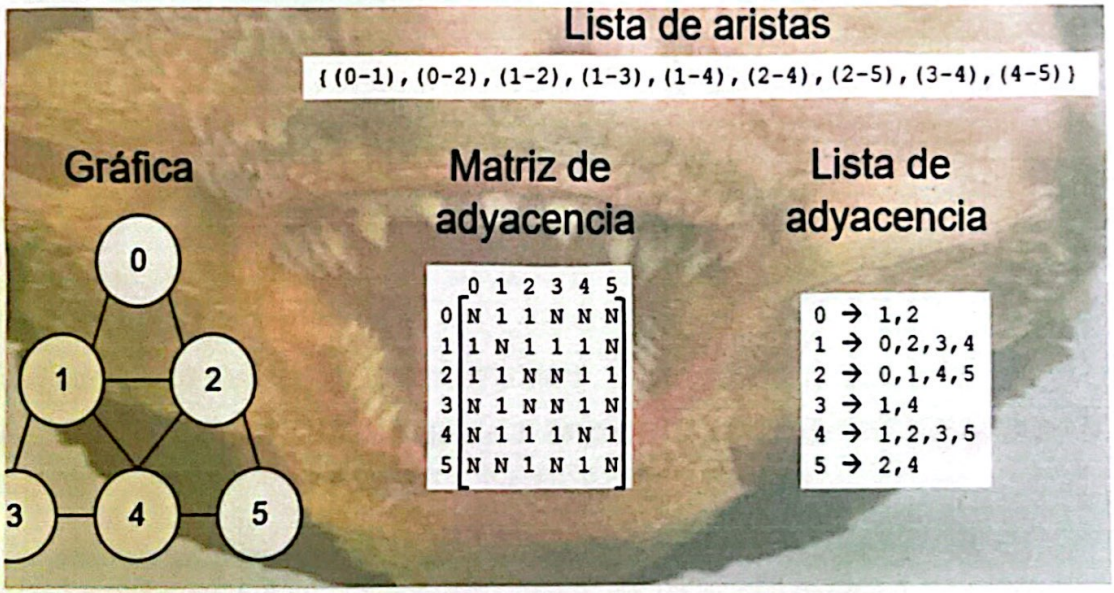
Open addressing -> una lista tan larga que el dato cae en un sitio que esta destinado. Hay un direccionamiento que esta abierto que depende del dato, super eficiente en búsqueda pero realmente en almacenamiento es un problema, la lista puede ser muy larga.

Grafos

Unión entre vértices(puntos) y aristas(rayas). G = (V,E)

Vértices: se refieren a aquellos puntos o nodos del grafo. Pueden contener datos o no.

Aristas: son las conexiones que comunican a un par de vértices. Pueden tener valores asociados (como costos) o no. Asimismo, las aristas pueden tener direccionalidad.



Todos representan es lo mismo

Si la matriz de adyacencia es simétrica el grafo es bidireccional, si solo uno cambia, ya no es bidireccional

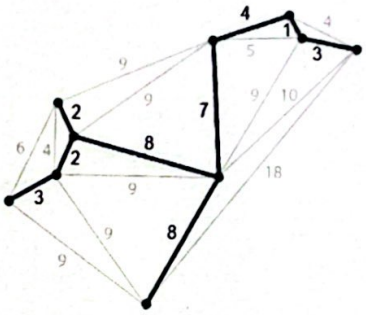
Aplicaciones: Los grafos se pueden usar como mapas que establecen localizaciones y rutas que las conectan (esto se puede asociar a transporte, aunque también tiene que ver con diseño de redes).

Algoritmos de Grafos

Encontrar arboles de cobertura mínima (PRIM y KRUSKAL)

Permite transitar de un vértice a cualquier otro del grafo, se construyen bajo grafos no dirigidos que tienen costos asociados a sus aristas.

//conecta a todos los vértices del grafo, pero con costo mínimo.



Adicionalmente, al tratarse de un árbol de cobertura mínimo, denota la configuración que cumple y es de menor costo.

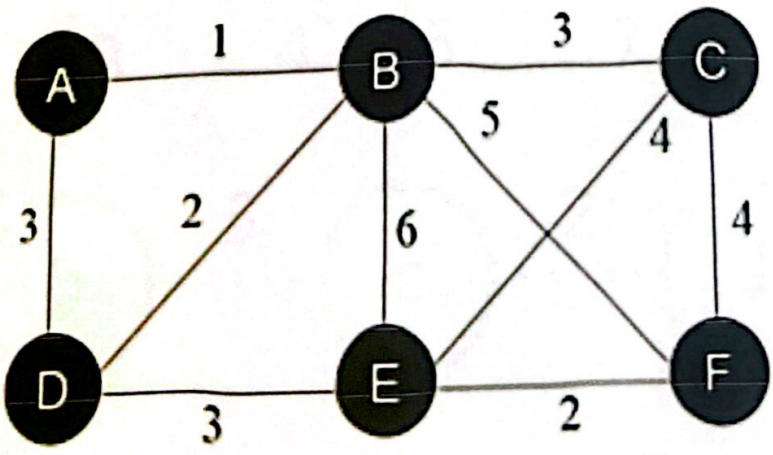
Grafos conexos → todos los nodos están conectados

Dos algoritmos conocidos son el de **PRIM** y el de **KRUSKAL** y ambos son de tipo codicioso (eligen la siguiente mejor opción durante el proceso de construcción del árbol)

Algoritmo de Prim (cobertura mínima)

- A partir de un grafo G:
  - Seleccionar un vértice v_1 al azar
  - Incluirlo en el conjunto de vértices visitados (ST)
  - Mientras no se hayan visitado todos los vértices de V:
    - Encontrar el conjunto de las aristas que conectan a algún vértice en ST con uno no-visitado (v_2)
    - Elegir la arista de menor costo e incluir a v_2 en ST

SET: { }



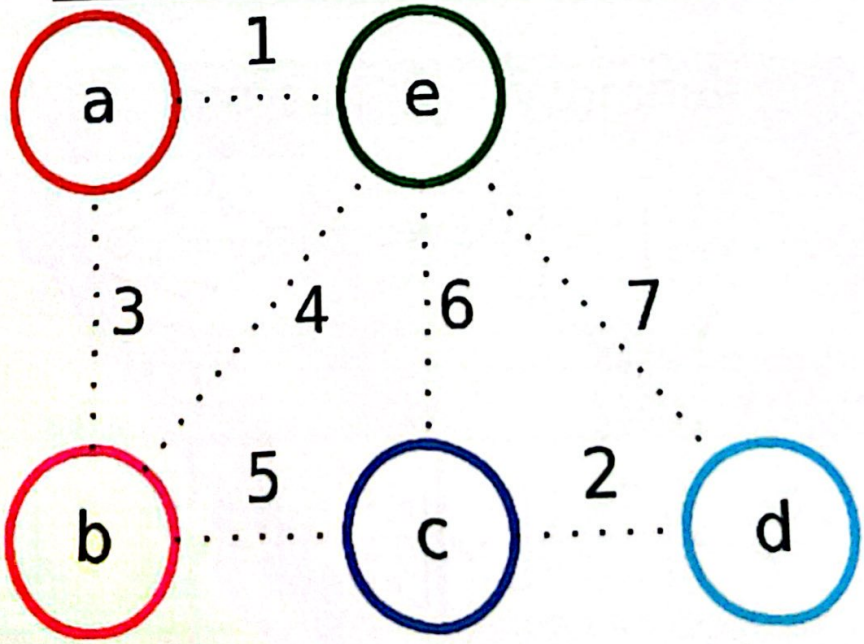
Algoritmo de Kruskal (cobertura mínima)

- A partir de un grafo G:
  - Ordenar el conjunto de las aristas E por costo
  - Seleccionar e incluir la arista de menor costo en el conjunto (e_1) de aristas presentes (ST) y registrar los vértices alcanzados
  - Mientras no se hayan alcanzado todos los vértices de V:
    - Seleccionar la siguiente arista de menor costo que no forme un ciclo (e_2)con las aristas en ST
    - Agregar e_2 a ST

No se tienen que formar ciclos, solo toma las aristas para comenzar, tiene que ser un grafo conexo

La parte del ciclo es un problema, se usa unión file ¿ fail ¿

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



RUTAS DE COSTO MINIMO

Se trata de un algoritmo se vean como elementos de trasporte y cada vértice es una localidad y la arista es el costo involucrado en desplazarse entre los vértices asociados

Existen 2 tipos de algoritmos de costo mínimo que se utilizan para los cálculos de rutas optimas:

Floyd Warshall (costo mínimo)

Se trata de un algoritmo de programación dinámica que calcula costos de las rutas óptimas entre todos los vértices del grafo (all-pairs shortest path).

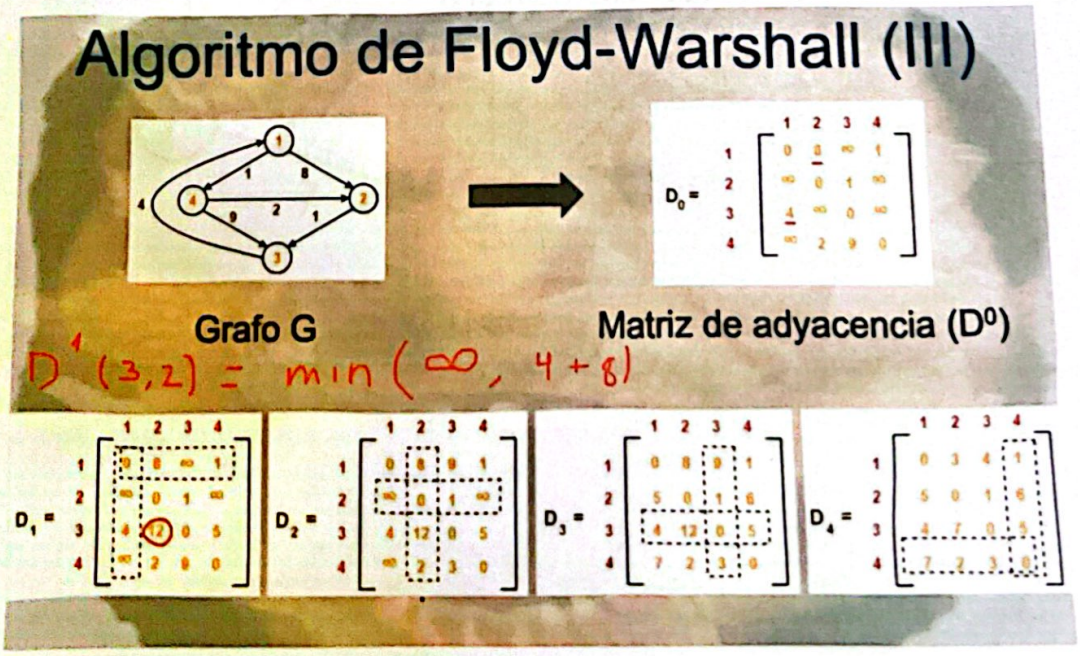
Es costoso, n cubo

Obtiene finalmente los costos de todas las mejores rutas entre cualquier par de vértices del grafo.

$D^k(i,j) = \min(D^{k-1}(i,j) , D^{k-1}(i,k) + D^{k-1}(k,j))$

// no es eficiente pero calcula todo

- Este algoritmo se basa en ir buscando todos los vértices como puntos intermedios en rutas entre dos vértices dados.
- Trabaja sobre grafos dirigidos.
- Obtiene, finalmente los costos de todas las mejores rutas entre cualquier par de vértices del grafo.



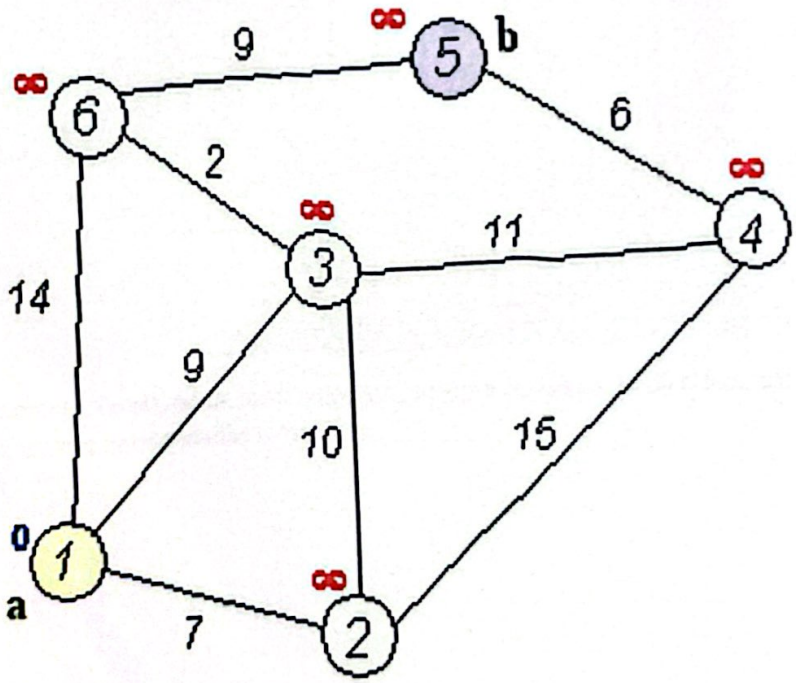
https://www.gatevidyalay.com/floyd-warshall-algorithm-shortest-path-algorithm/

código y ejemplo

Dijkstra (costo mínimo)

es un algoritmo codicioso que calcula costos de la ruta óptima entre un vértice y todos los otros del grafo.

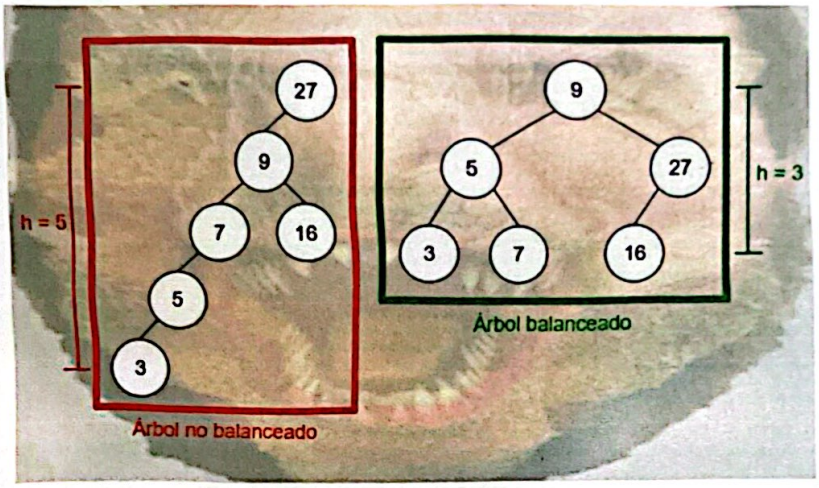
- Calcula los costos de todas las rutas mínimas desde un vértice de origen a todos los otros vértices en un grafo G.
- No admite aristas de costo negativo.
- Trabaja sobre grafos dirigidos.



https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra

ARBOLES AVL

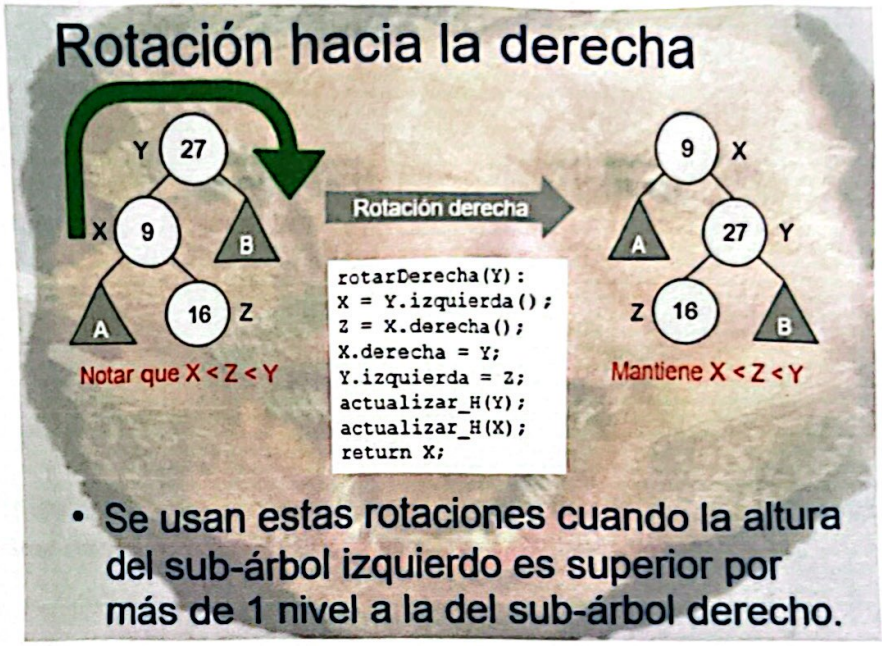
- Sabemos que los árboles binarios tienen la característica de potencialmente lograr operaciones en un tiempo de O(log n), si n es el número de elementos en el árbol.
- Sin embargo, esa complejidad no es garantizada, pues en casos malos podría llegar a ser O(n)
- Árbol binario (dos hijos) que se auto balancea

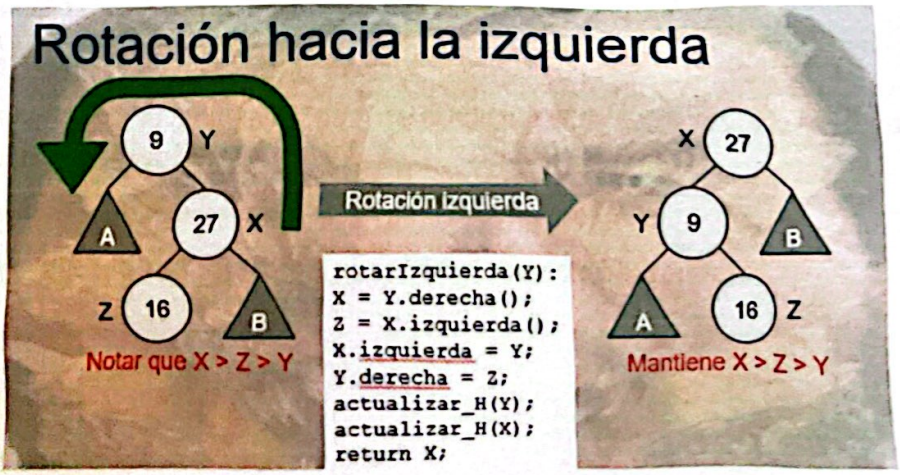


- Efectivamente. En el caso de los árboles AVL, se logra converger hacia el balanceo por medio de operaciones denominadas rotaciones.

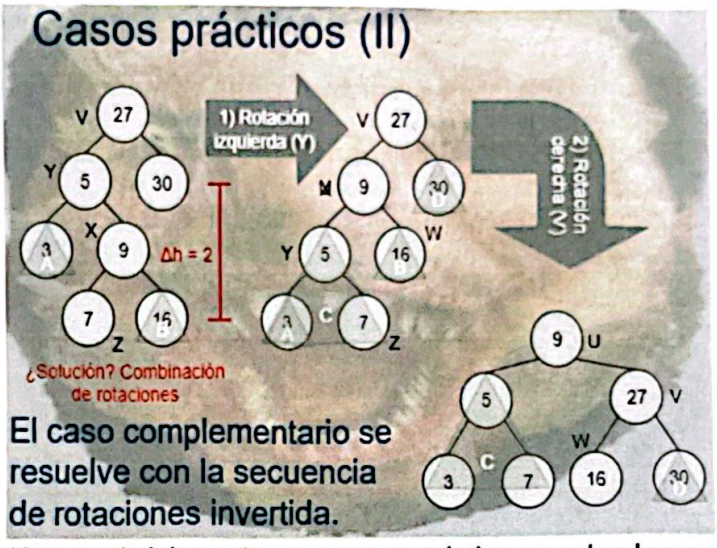
Rotaciones

Puede haber rotaciones a la derecha o a la izquierda





Rotaciones dobles:



Inserción: Es igual al árbol binario de búsqueda, pero usa un rebalanceo sobre la rama que se colocó el nuevo nodo.

Borrado: Para borrar el nodo primero se necesita buscarlo y encontrarlo:

- Si es una hoja, no se reemplaza.
- Si tiene un solo hijo, se reemplaza por ese hijo.
- Si tiene 2 hijos, se reemplaza por el descendiente derecho de mas a la izquierda y recursivamente se borra.
- Finalmente se balancea hacia arriba para preservar el AVL.

Búsqueda: se realiza de la misma forma que el árbol vinario de búsqueda:

Árbol Heap:

- Árbol binario aplanado a un arreglo
- Los hijos del elemento i son elementos 2i+1 y 2i+2 (relación entre los valores)
- Max-heap: nodo padre mayor que sus 2 hijos, se cumple en sub-arboles
- Min-heap: nodo padre es menor que sus 2 hijos, se cumple para sub-arboles
- Es una implementación eficiente de colas priorizadas

Algoritmos de Ordenamiento

•**BubbleSort** → (n cuadrado)

•**InsertSort** → peor caso es que el arreglo este al revés (n cuadrado)

•**HeapSort** → uno de los mejores algoritmos de ordenamiento,

•**MergeSort** → otro de los buenos algoritmos, pedazos, por secciones se trabajan (n log n)

•**QuickSort** → (n log n) tomar un pivote y alrededor todo lo de la derecha es mayor y todo lo de la izquierda es menor...

Algoritmo	Detalle	O	θ	Ω
BubbleSort	Por cada	$O(n^2)$	$\theta(n^2)$	$\Omega(n^2)$
InsertSort	dad	$O(n^2)$	$O(n^2)$	$\Omega(n)$
QuickSort	SE ocupa un	$O(n^2)$	$\theta(n \log(n))$	$\Omega(n \log(n))$
MergeSort	Corta en 2 hasta que quede lo mas pequeño y hace Merge para ordenar.	$O(n \log(n))$	$\theta(n \log(n))$	$\Omega(n \log(n))$
HeapSort	dad	$O(n \log(n))$	$\theta(n \log(n))$	$\Omega(n \log(n))$

Clases de Algoritmos

según formatos se pueden organizar:

- **Divide and conquer** → dividir un problema para después resolverlo, merge sort es un ejemplo

- **Greedy** → algoritmo codicioso, encontrar una solución parcial , avanzando en la solución, esta solución parcial es la mejor que puedo ver en el momento (no necesariamente es la mejor a largo plazo)

  Un ejemplo es el ajedrez, un paso puede ser comer una figura pero puede ser inducida por el otro jugador

- Dynamic programming → Recordar ciertas soluciones que se han resuelto antes para acelerar el proceso de solución completa.

Recorridos y búsquedas

- **BFS →** búsqueda en anchura, amplitud, si revisa todos por mas lento que sea, cae en todos los vértices
- **DFS →** Búsqueda en profundidad, hay problemas cuando hay bidireccionalidad, puede caer en un loop.
- **Backtracking →** volver al ultimo punto donde tenía una opción y cambiar asa opción.

Algoritmo exhaustivo: revisa todas las posibilidades