https://www.bigocheatsheet.com/

BFS uses always **queue**, Dfs uses **Stack data structure**. As the earlier explanation tell about DFS is using backtracking. Remember backtracking can proceed only by **Stack**. The depth-first search uses a **Stack** to remember where it should go when it reaches a dead end.

BFS uses Queue to find the shortest path. **DFS** uses Stack to find the shortest path. **BFS** is **better** when target is closer to Source. **DFS** is **better** when target is far from source

Dijkstra's Algorithm is a **greedy algorithm**. ... **Dijkstra's**, as most of us know, is an **algorithm** which finds the shortest path from a source/node. Similar to Prim's **algorithm** to find the minimum spanning tree, we always choose the most optimal local solution.

It differs from minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

Heap is **generally preferred for priority queue** implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, getHighestPriority() can be implemented in O(1) time, insert() can be implemented in O(Logn) time and deleteHighestPriority() can also be implemented in O(Logn) time.

Balanced Tree

1. A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1.
2. A binary tree is balanced if for any two leaves the difference of the depth is at most 1.

There exist over one hundred types of **balanced search trees**. Some of the more common types are: **AVL trees**, **B-trees**, and **red-black** trees

AVL

an AVL tree is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one

AVL tree is a self-balancing binary search tree in which each node maintains an extra information called as balance factor whose value is either -1, 0 or +1

RBL

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

1) Every node has a color either red or black.
2) Root of tree is always black.
3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).

B-Tree

A **B-tree** is a **tree** data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search **trees**, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems

Hash Table

a hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found

HashMap

HashMap and Hashtable store key/value pairs in a hash table. When using a Hashtable or HashMap, we specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.a

Hash table lookups are O(1) in the average case, and O(n) in the worst case.

Hash tables are just one implementation of associative arrays. Associative arrays (also known as dictionaries or maps) are the abstract data structure mapping keys to values. Implementations of associative arrays using self-balancing binary search trees have lookup that is O(log n) in the worst case.

What is the lower bound for number of comparisons while sorting ?

(omega)

Ω(n log(n)) **comparisons** to distinguish between the n ! possible permutations of n distinct **numbers**. This means that Ω(n log(n)) is a **lower bound** for the time complexity of any **sorting** algorithm that is based on **comparisons**

A sorting algorithm that is based on comparisons draws exactly 1 bit of information from each comparison (namely whether the two numbers compared are in order or not). Thus it needs at least $\log(n!) \in \Omega(n \log(n))$ comparisons to distinguish between the $n!$ possible permutations of n distinct numbers. This means that $\Omega(n \log(n))$ is a lower bound for the time complexity of any sorting algorithm that is based on comparisons.

The sorting algorithms Heapsort und Mergesort have an upper bound of $O(n \log(n))$ steps. Therefore, they are optimal since they attain the lower bound.

There are sorting algorithms that are not based on comparisons, e.g. Bucket Sort and Radix Sort. These algorithms draw more than 1 bit of information from each step. Therefore, the information theoretic lower bound is not likewise a lower bound for the time complexity of these algorithms. In fact, Bucket Sort and Radix Sort have a time complexity of $O(n)$. However, these algorithms are not as general as comparison based algorithms since they rely on certain assumptions concerning the data to be sorted.

Binary Search is a searching algorithm for finding an element's position in a sorted array.

In this approach, the element is always searched in the middle of a portion of an array.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

Let’s see the difference between B-tree and Binary tree:

S.NO	B-TREE	BINARY TREE
1.	In a B-tree, a node can have maximum ‘M'(‘M’ is the order of the tree) number of child nodes.	While in binary tree, a node can have maximum two child nodes or sub-trees.
2.	B-tree is called as sorted tree as its nodes are sorted in inorder traversal.	While binary tree is not a sorted tree. It can be sorted in inorder, preorder or postorder traversal.
3.	B-tree has a height of logM N (Where ‘M’ is the order of tree and N is the number of nodes).	While binary tree has a height of log2 N(Where N is the number of nodes).
4.	B-Tree is performed when the data is loaded in the disk.	Unlike B-tree, binary tree is performed when the data is loaded in the RAM(faster memory).
5.	B-tree is used in DBMS(code indexing, etc).	While binary tree is used in Huffman coding and Code optimization and many others.
6.	To insert the data or key in B-tree is more complicated than binary tree.	While in binary tree, data insertion is not complicated than B-tree.

2^24 = 16777216 (1024*1024 *16)

How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum.
We start from the edges with the lowest weight and keep adding edges until we we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.

Floyd-Warshall

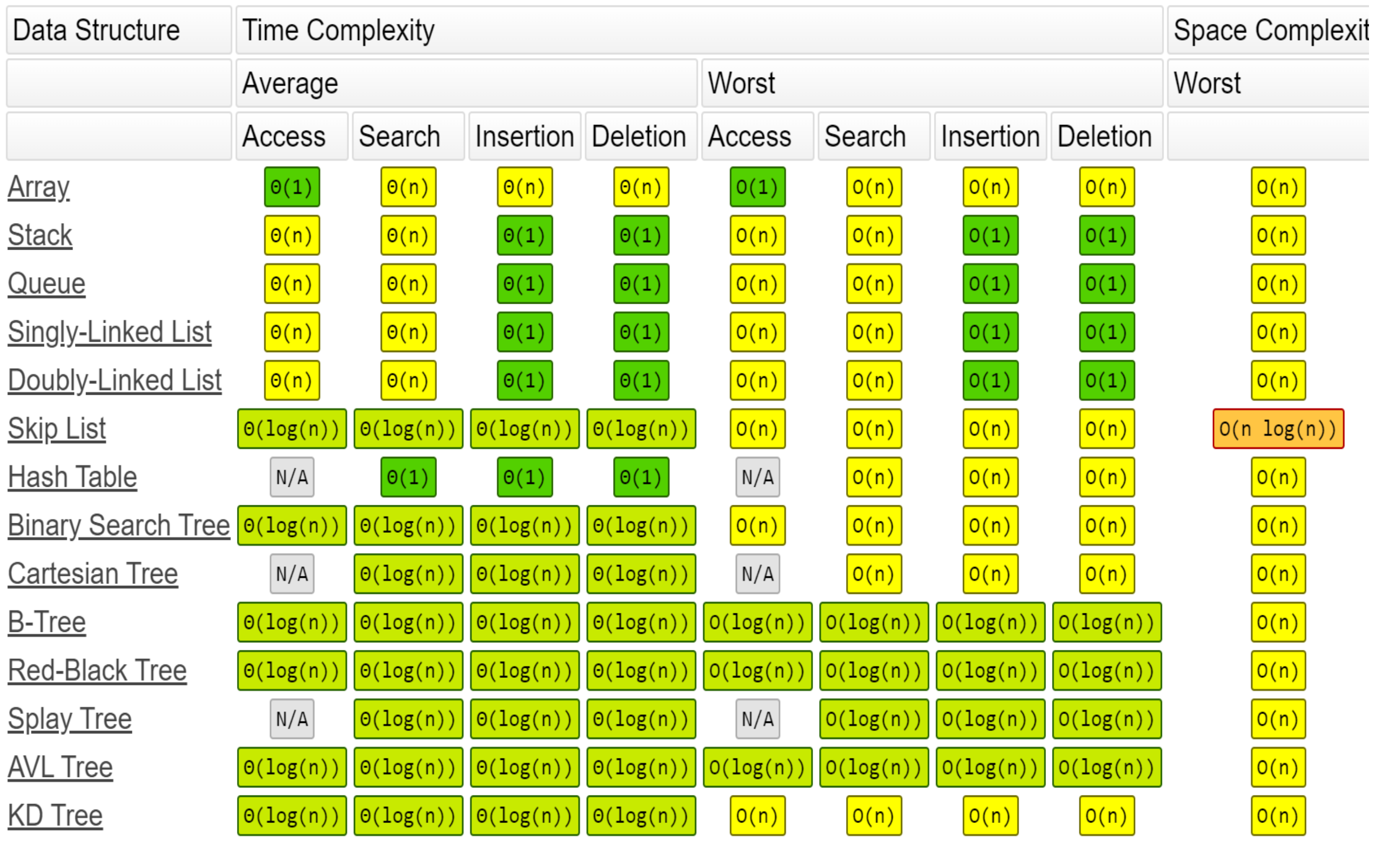
Matrix

Longest Common Subsequence

Create a table of dimension `n+1*m+1`

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)



Data Structure	Time Complexity								Space Complexit
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)