"Career With Rishab" Complexity Cheat Sheet					

Complexity of Different Data Structures

Data Structure	Operation	Time Complexity			
		Worst Case	Average Case	Best Case	Explanation
Array	Access	O(1)	O(1)	O(1)	Accessing an element in an array by index takes constant time as the location can be calculated directly.
	Search	O(n)	O(n)	O(1)	In the worst case, searching an unsorted array requires traversing through all elements. In the best case, if the element is at the beginning, it takes constant time.
	Insertion	O(n)	O(n)	O(1)	In the worst case, inserting an element at the beginning of an array requires shifting all other elements.
	Deletion	O(n)	O(n)	O(1)	In the worst case, deleting an element from the beginning of an array requires shifting all other elements.
Singly Linked List	Access	O(n)	O(n)	O(1)	Accessing an element in a linked list requires traversing the list from the head to the desired element.
	Search	O(n)	O(n)	O(1)	Searching in a linked list involves traversing the entire list to find the desired element.
	Insertion	O(1)	O(1)	O(1)	Inserting an element at the head or tail of a linked list can be done in constant time by updating the necessary pointers.
	Deletion	O(1)	O(1)	O(1)	Deleting the head or tail of a linked list can be done in constant time by updating the necessary pointers.
Doubly Linked List	Access	O(n)	O(n)	O(1)	Accessing an element in a doubly linked list requires traversing the list from the head or tail to the desired element.
	Search	O(n)	O(n)	O(1)	Searching in a doubly linked list involves traversing the entire list to find the desired element.
	Insertion	O(1)	O(1)	O(1)	Inserting an element at the head or tail of a doubly linked list can be done in constant time by updating the necessary pointers.
	Deletion	O(1)	O(1)	O(1)	Deleting a node from a doubly linked list can be done in constant time by updating the necessary pointers.
Stack	Push	O(1)	O(1)	O(1)	Pushing an element onto a stack takes constant time as it involves updating the top of the stack.
	Pop	O(1)	O(1)	O(1)	Popping an element from a stack takes constant time as it involves updating the top of the stack.
	Peek	O(1)	O(1)	O(1)	Peeking at the top element of a stack takes constant time as it does not modify the stack.
Queue	Enqueue	O(1)	O(1)	O(1)	Enqueueing an element into a queue takes constant time as it involves updating the tail of the queue.
	Dequeue	O(1)	O(1)	O(1)	Dequeuing an element from a queue takes constant time as it involves updating the head of the queue.
	Peek	O(1)	O(1)	O(1)	Peeking at the front element of a queue takes constant time as it does not modify the queue.
Heap	Insert	O(log n)	O(log n)	O(1)	Inserting an element into a heap takes logarithmic time as it may require restructuring the heap.
	Delete	O(log n)	O(log n)	O(1)	Deleting the minimum or maximum element from a heap takes logarithmic time as it may require restructuring the heap.
	Get Min/Max	O(1)	O(1)	O(1)	Retrieving the minimum or maximum element from a heap can be done in constant time.
Binary Search Tree	Search	O(n)	O(log n)	O(1)	In the worst case, a binary search tree can be a skewed binary search tree and can lead to linear time complexity for searching.
	Insert	O(n)	O(log n)	O(log n)	In the worst case, inserting elements in a binary search tree without balancing can lead to linear time complexity.
	Delete	O(n)	O(log n)	O(log n)	In the worst case, deleting elements in a binary search tree without balancing can lead to linear time complexity.
AVL Tree	Search	O(log n)	O(log n)	O(1)	Searching in an AVL tree takes logarithmic time as it maintains balanced height through rotations.
	Insert	O(log n)	O(log n)	O(log n)	Inserting an element into an AVL tree requires rotations to maintain balance, resulting in logarithmic time complexity.
	Delete	O(log n)	O(log n)	O(log n)	Deleting an element from an AVL tree requires rotations to maintain balance, resulting in logarithmic time complexity.
Red-Black Tree	Search	O(log n)	O(log n)	O(log n)	Searching in a red-black tree takes logarithmic time as it maintains balanced height and adheres to the properties of a binary search tree.
	Insert	O(log n)	O(log n)	O(log n)	Inserting an element into a red-black tree requires restructuring to maintain balance, resulting in logarithmic time complexity.
	Delete	O(log n)	O(log n)	O(log n)	Deleting an element from a red-black tree requires restructuring to maintain balance, resulting in logarithmic time complexity.
Hash Table	Search	O(1)	O(1)	O(1)	Hash tables provide constant-time search on average, assuming a good hash function and proper handling of collisions.
	Insert	O(1)	O(1)	O(1)	Inserting into a hash table takes constant time on average, assuming a good hash function and proper handling of collisions.
	Delete	O(1)	O(1)	O(1)	Deleting from a hash table takes constant time on average, assuming a good hash function and proper handling of collisions.

n = number of elements

Complexity of Different Array Algorithms

Array Algorithm	Time Complexity				Space Complexity	
	Worst Case	Average Case	Best Case	Explanation	Worst Case	Explanation
Linear Search	O(n)	O(n)	O(1)	Iterate through the array	O(1)	Only need constant space for variables
Binary Search	O(log n)	O(log n)	O(1)	Divide and conquer approach	O(1)	Only need constant space for variables
Ternary Search	O(log3 n)	O(log3 n)	O(1)	Divide and conquer approach	O(1)	Only need constant space for variables
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Divide and conquer approach	O(n)	Requires additional space for merging
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Find the minimum element	O(1)	Only need constant space for variables
Bubble Sort	O(n^2)	O(n^2)	O(n)	Compare adjacent elements	O(1)	Only need constant space for variables
Insertion Sort	O(n^2)	O(n^2)	O(n)	Insert element in sorted order	O(1)	Only need constant space for variables
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Build max/min heap and extract elements	O(1)	Only need constant space for variables
Quick Sort	O(n^2)	O(n log n)	O(n log n)	Divide and conquer approach	O(log n)	Requires space for the recursion stack
Randomized Quick Sort	O(n^2)	O(n log n)	O(n log n)	Randomly select pivot	O(log n)	Requires space for the recursion stack
Bucket Sort	O(n^2)	O(n + k)	O(n + k)	Distribute elements into buckets	O(n + k)	Requires additional space for buckets
Counting Sort	O(n+k)	O(n+k)	O(n+k)	Count occurrences of elements	O(n+k)	Requires additional space for counting
Kadane's Algorithm	O(n)	O(n)	O(n)	Find maximum subarray sum	O(1)	Only need constant space for variables

Complexity of Different Graph Algorithms

Graph Algorithm	Time Complexity				Space Complexity	
	Worst Case	Average Case	Best Case	Explanation	Worst Case	Explanation
Depth First Search	O(V + E)	O(V + E)	O(V + E)	Traverse all vertices and edges in a graph	O(V)	Requires space for the recursion stack
Breath First Search	O(V + E)	O(V + E)	O(V + E)	Traverse all vertices and edges in a graph	O(V)	Requires space for the queue
Dijkstra's Algorithm	O(V^2)	O((V+E) log V)	O(V log V)	Find shortest path from a source to all other vertices in a graph.	O(V)	Requires space for the priority queue
Floyd-Warshall Algorithm	O(V^3)	O(V^3)	O(V^3)	Find shortest path between all pairs of vertices in a graph	O(V^2)	Requires space for the distance matrix
Prim's Algorithm	O((V + E) log V)	O((V + E) log V)	O((V + E) log V)	Construct a minimum spanning tree in a graph	O(V)	Requires space for the priority queue
Kruskal's Algorithm	O(E log E)	O(E log E)	O(E log E)	Construct a minimum spanning tree in a graph	O(V)	Requires space for the disjoint set data structure
Bellman-Ford Algorithm	O(VE)	O(VE)	O(VE)	Find the shortest path from a source to all other vertices in a graph	O(V)	Requires space for the distance array
0/1 Knapsack Problem	O(NW)	O(NW)	O(NW)	Find the maximum value subset from a given set of items	O(W)	Requires space for the memoization table
Topological Sorting	O(V + E)	O(V + E)	O(V + E)	Sort the vertices in a directed acyclic graph	O(V)	Requires space for the stack

V = Number of Vertices
E = Number of Edges
W = Capacity of the Knapsack
N = Number of items available