Time Complexity:

Time complexity refers to the amount of time an algorithm takes to run as a function of the input size. It measures the number of operations performed by the algorithm relative to the size of the input. Time complexity is also expressed using Big O notation.

Example: Let's take the example of searching for an element in an unsorted array. If you use linear search, the time taken will be directly proportional to the size of the array. Therefore, the time complexity of linear search is O(n), where n is the size of the array. On the other hand, if you use binary search on a sorted array, the time complexity is O(log n), where n is the size of the array. This means that as the size of the input grows, the time taken to perform the binary search grows logarithmically.

Time complexcity

1) Big O (O): This notation is used to describe the upper bound of an algorithm's time complexity. It represents the maximum amount of time an algorithm takes to complete, given any input size. For example, if an algorithm has a time complexity of O(n), it means that the algorithm's running time grows linearly with the size of the input.

Big O (O): Upper Bound

f(n) is O (g (n)) if there exist constants c and n _0 such that 0 ≤ f (n) ≤ c g (n) for all n ≥ n_0

f(n)=O(g(n))

2) Big Omega (Ω): This notation represents the lower bound of an algorithm's time complexity. It describes the best-case scenario of an algorithm's running time. For example, if an algorithm has a time complexity of Ω(n), it means that the algorithm will take at least linear time to complete.

Big Omega (Ω): Lower Bound

f(n) is Ω(g(n)) if there exist constants c and n_0such that 0≤cg(n)≤f(n) for all n≥n_0.

f(n)=Ω(g(n))

3) Big Theta (Θ): This notation provides a tight bound on an algorithm's time complexity. It represents both the upper and lower bounds of the algorithm's running time. For example, if an algorithm has a time complexity of Θ(n), it means that the algorithm's running time grows linearly with the size of the input, and it cannot be worse or better than linear time.

Big Theta (Θ): Tight Bound

f(n) is Θ(g(n)) if there exist constants c 1 , c 2, and n_ 0such that 0≤c1g(n)≤f(n)≤c2g(n) for all n≥n_0.

f(n)=Θ(g(n))

4) Small O (o): This notation describes an upper bound on an algorithm's time complexity that is not tight. It means that the algorithm's running time grows strictly slower than the specified function. For example, if an algorithm has a time complexity of o(n), it means that the algorithm's running time grows slower than linear time.

Small O (o): Upper Bound (Not Tight)

f(n) is o(g(n)) if for any constant c>0, there exists a constant n_0 such that 0≤f(n)<cg(n) for all n≥n_0.

f(n)=o(g(n))

5) Small Theta (θ): This notation describes a tight bound on an algorithm's time complexity, similar to Big Theta. However, it's used for average-case analysis rather than worst-case analysis. It represents both the upper and lower bounds of an algorithm's running time on average.

Small Theta (θ): Average Case Bound (Tight)

f(n) is θ(g(n)) if there exist constants c 1 , c 2 , and n_0 such that 0≤c1g(n)≤f(n)≤c2g(n) for infinitely many values of n.

f(n)=θ(g(n))

1. O(1): Constant time complexity. This means that the algorithm's runtime is not dependent on the size of the input. Examples include accessing an element in an array by index or performing basic arithmetic operations. O(1): Constant time complexity.

Example: Accessing an element in an array by index. It doesn't matter how large the array is; accessing an element takes the same amount of time

2. O(log N): Logarithmic time complexity. Algorithms with this complexity typically divide the problem in each step. Examples include binary search in a sorted array or certain tree operations like searching in a balanced binary search tree (BST). O(log N): Logarithmic time complexity.

Example: Binary search in a sorted array. Each step of the binary search reduces the search space by half, making it highly efficient even for large datasets.

3. O(N): Linear time complexity. The runtime of the algorithm grows linearly with the size of the input. Examples include iterating through an array or list to find a specific element. O(N): Linear time complexity.

Example: Finding an element in an unsorted array by iterating through it. As the array grows, the time taken to find the element grows linearly.

4. O(N log N): Linearithmic time complexity. This is often seen in efficient sorting algorithms like Merge Sort, Quick Sort, or Heap Sort. O(N log N): Linearithmic time complexity.

Example: Merge Sort. In Merge Sort, the array is divided into halves recursively until single elements remain, then merged back together in sorted order. The time complexity arises from the merging step.

5. O(N^2): Quadratic time complexity. Algorithms with this complexity often involve nested iterations over the input data. Examples include certain implementations of sorting algorithms like Bubble Sort or algorithms for certain types of dynamic programming problems. O(N^2): Quadratic time complexity.

Example: Bubble Sort. In Bubble Sort, each element is compared with its adjacent element and swapped if they are in the wrong order. This process is repeated for each element, resulting in a time complexity of O(N^2).

6. O(N^k): Polynomial time complexity. Where k is a constant, this represents algorithms whose runtime grows with the input size raised to a constant power. Examples include certain matrix operations. O(N^k): Polynomial time complexity.

Example: Matrix multiplication. For two matrices of size N x N, the naive algorithm requires N^3 operations, resulting in a time complexity of O(N^3).

7. O(2^N): Exponential time complexity. Algorithms with this complexity often involve generating all possible combinations or permutations of the input. Examples include the naive recursive solution for the traveling salesman problem or certain recursive algorithms for generating subsets.

O(2^N): Exponential time complexity.

Example: The naive recursive solution for the Traveling Salesman Problem. This problem involves finding the shortest possible route that visits every city exactly once and returns to the origin city. The naive recursive solution explores all possible permutations of cities, resulting in an exponential time complexity.

8. O(N!): Factorial time complexity. Algorithms with this complexity are highly inefficient and typically involve generating all possible permutations of the input. Examples include the naive recursive solution for problems like finding all permutations of a set. O(N!): Factorial time complexity.

Example: Finding all permutations of a set. This involves generating all possible arrangements of the elements in the set, resulting in a time complexity of O(N!), where N is the number of elements in the set.

When comparing these complexities, generally, we say that O(log N) is more efficient than O(N) for large inputs, O(N) is more efficient than O(N log N), and so on, with O(1) being the most efficient

$$O(1)<O(logN)<O(N)<O(NlogN)<O(N^2)<O(2^N)<O(N!)$$

Space Complexity:

Space complexity refers to the amount of memory space required by an algorithm to solve a problem as a function of the input size. It quantifies the resources (usually memory) needed by an algorithm to complete its execution. Space complexity is often measured in terms of Big O notation.

Example: Consider a program that sorts an array of numbers using the Bubble Sort algorithm. In this algorithm, no extra space is required other than the input array itself. However, if you implement Merge Sort, it may require additional space for temporary arrays during the sorting process. The space complexity of Bubble Sort is O(1) (constant space), while the space complexity of Merge Sort is O(n) (linear space), where n is the size of the input array.

SPACE COMPLEXCITY 1)O(1): Constant space complexity. This means that the amount of memory used by the algorithm remains constant, regardless of the size of the input. Examples include variables with fixed size or a fixed number of memory allocations.

2) O(log N): Logarithmic space complexity. Algorithms with this complexity typically use a space that grows logarithmically with the size of the input. Examples include certain recursive algorithms where the space used decreases with each recursive call or algorithms that use a constant amount of additional space per level of recursion.

3) O(N): Linear space complexity. The amount of memory used by the algorithm grows linearly with the size of the input. Examples include algorithms that store the entire input or data structures that scale proportionally with the input size.

4) O(N log N): Linearithmic space complexity. This complexity often accompanies algorithms that use divide-and-conquer strategies and require additional space for recursive calls or data structures. Examples include merge sort or certain tree traversal algorithms.

5) O(N^2): Quadratic space complexity. The amount of memory used grows quadratically with the size of the input. Examples include algorithms that use a two-dimensional array or nested data structures with a size proportional to the square of the input size.

6) O(N^k): Polynomial space complexity. Similar to polynomial time complexity, this represents algorithms whose memory usage grows with the input size raised to a constant power. Examples include certain matrix operations or algorithms that use nested loops with fixed-size data structures.

7) O(2^N): Exponential space complexity. The amount of memory used grows exponentially with the size of the input. Examples include algorithms that generate all possible combinations or subsets of the input, where each additional element doubles the space required.

8) O(N!): Factorial space complexity. Algorithms with this complexity use a factorial amount of memory, which quickly becomes impractical for large inputs. Examples include algorithms that generate all permutations or combinations of the input, requiring storage for every possible arrangement.

In terms of efficiency, just like with time complexity, lower space complexities are generally more desirable. Algorithms with O(1) space complexity are the most efficient in terms of memory usage, followed by those with logarithmic, linear, and so on

1. array :

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Accessing an Element	O(1)	O(1)	O(1)	Accessing an element in an array requires constant time.	O(1)	O(1)	O(1)	Memory required to store the element is constant.
Insertion at End (Append)	O(1)	O(1) amortized	O(n)	If there's available space, insertion is constant time.	O(1)	O(1) amortized	O(n)	Resizing may require allocating a larger array.
Insertion at Beginning/Middle	O(n)	O(n)	O(n)	Requires shifting elements, which takes linear time.	O(1)	O(1)	O(n)	Resizing may require allocating a larger array.
Deletion at End	O(1)	O(1)	O(1)	Deleting from the end requires updating array length.	O(1)	O(1)	O(1)	Memory required remains constant.
Deletion at Beginning/Middle	O(n)	O(n)	O(n)	Requires shifting elements, which takes linear time.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying requires iterating through all elements.	O(n)	O(n)	O(n)	Memory required to store copied elements is linear.

2. string operations:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Accessing a Character	O(1)	O(1)	O(1)	Accessing a character in a string requires constant time.	O(1)	O(1)	O(1)	Memory required to store the character is constant.
Concatenation	O(n)	O(n)	O(n)	Concatenating two strings requires iterating through all characters.	O(n)	O(n)	O(n)	Memory required to store the concatenated string is linear.
Substring	O(1)	O(n)	O(n)	Extracting a substring requires constant time for fixed-length substrings, but linear time for variable-length substrings.	O(1)	O(n)	O(n)	Memory required to store the substring is linear.
Searching (Linear Search)	O(n)	O(n)	O(n)	Searching for a substring using linear search requires iterating through the entire string.	O(1)	O(1)	O(1)	Memory required remains constant.
Searching (Pattern Matching Algorithms)	O(n + m)	O(n + m)	O(n * m)	Pattern matching algorithms like Knuth-Morris-Pratt or Boyer-Moore have different complexities depending on the algorithm and input.	O(1)	O(1)	O(1)	Memory required remains constant.
Reversal	O(n)	O(n)	O(n)	Reversing a string requires iterating through all characters.	O(n)	O(n)	O(n)	Memory required to store the

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
								reversed string is linear.

3. linked list operations:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Accessing an Element	O(1)	O(n)	O(n)	Accessing an element in a linked list by index requires traversing the list.	O(1)	O(1)	O(1)	Memory required to store the accessed element is constant.
Insertion at Beginning	O(1)	O(1)	O(1)	Inserting at the beginning requires updating pointers.	O(1)	O(1)	O(1)	Memory required to store the new node is constant.
Insertion at End	O(1)	O(1)	O(1)	Inserting at the end requires updating pointers.	O(1)	O(1)	O(1)	Memory required to store the new node is constant.
Insertion in Middle	O(1)	O(n)	O(n)	Inserting in the middle requires traversing to the insertion point.	O(1)	O(1)	O(1)	Memory required to store the new node is constant.
Deletion at Beginning	O(1)	O(1)	O(1)	Deleting from the beginning requires updating pointers.	O(1)	O(1)	O(1)	Memory required remains constant.
Deletion at End	O(1)	O(n)	O(n)	Deleting from the end requires traversing to the end of the list.	O(1)	O(1)	O(1)	Memory required remains constant.
Deletion in Middle	O(1)	O(n)	O(n)	Deleting in the middle requires traversing to the deletion point.	O(1)	O(1)	O(1)	Memory required remains constant.
Searching (Linear Search)	O(n)	O(n)	O(n)	Searching for an element using linear search requires traversing the list.	O(1)	O(1)	O(1)	Memory required remains constant.
Reversal	O(n)	O(n)	O(n)	Reversing a linked list requires iterating through all nodes.	O(n)	O(n)	O(n)	Memory required to store the reversed list is linear.
Copying	O(n)	O(n)	O(n)	Copying a linked list requires iterating through all nodes.	O(n)	O(n)	O(n)	Memory required to store the copied list is linear.

4. stack operations:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Push	O(1)	O(1)	O(1)	Pushing an element onto a stack requires constant time.	O(1)	O(1)	O(1)	Memory required to store the pushed element is constant.
Pop	O(1)	O(1)	O(1)	Popping an element from a stack requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Peek (Top)	O(1)	O(1)	O(1)	Retrieving the top element of a stack requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.
Search	O(n)	O(n)	O(n)	Searching for an element in a stack requires linear time.	O(1)	O(1)	O(1)	Memory required remains constant.
Size	O(1)	O(1)	O(1)	Determining the size of a stack requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying a stack requires iterating through all elements.	O(n)	O(n)	O(n)	Memory required to store the copied stack is linear.

5. queue operations:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Enqueue	O(1)	O(1)	O(1)	Enqueuing an element into a queue requires constant time.	O(1)	O(1)	O(1)	Memory required to store the enqueued element is constant.
Dequeue	O(1)	O(1)	O(1)	Dequeuing an element from a queue requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.
Peek (Front)	O(1)	O(1)	O(1)	Retrieving the front element of a queue requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.
Search	O(n)	O(n)	O(n)	Searching for an element in a queue requires linear time.	O(1)	O(1)	O(1)	Memory required remains constant.
Size	O(1)	O(1)	O(1)	Determining the size of a queue requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying a queue requires iterating through all elements.	O(n)	O(n)	O(n)	Memory required to store the copied queue is linear.

6. binary tree operations:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Search	O(1)	O(log n)	O(n)	Searching for an element in a binary tree.	O(1)	O(1)	O(1)	Memory required remains constant. (Assuming no additional data structure)
Insertion	O(1)	O(log n)	O(n)	Inserting an element into a binary tree.	O(1)	O(1)	O(1)	Memory required remains constant. (Assuming no additional data structure)
Deletion	O(1)	O(log n)	O(n)	Deleting an element from a	O(1)	O(1)	O(1)	Memory required remains constant.

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
				binary tree.				(Assuming no additional data structure)
Traversal (Inorder, Preorder, Postorder)	O(n)	O(n)	O(n)	Traversing all nodes of a binary tree.	O(n)	O(n)	O(n)	Memory required to store the traversal result is linear.
Height	O(1)	O(log n)	O(n)	Determining the height of a binary tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying a binary tree.	O(n)	O(n)	O(n)	Memory required to store the copied binary tree is linear.

7. binary search tree (BST) operations:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Search	O(1)	O(log n)	O(n)	Searching for an element in a binary search tree.	O(1)	O(1)	O(1)	Memory required remains constant. (Assuming no additional data structure)
Insertion	O(1)	O(log n)	O(n)	Inserting an element into a binary search tree.	O(1)	O(1)	O(1)	Memory required remains constant. (Assuming no additional data structure)
Deletion	O(1)	O(log n)	O(n)	Deleting an element from a binary search tree.	O(1)	O(1)	O(1)	Memory required remains constant. (Assuming no additional data structure)
Traversal (Inorder, Preorder, Postorder)	O(n)	O(n)	O(n)	Traversing all nodes of a binary search tree.	O(n)	O(n)	O(n)	Memory required to store the traversal result is linear.
Minimum/Maximum	O(1)	O(log n)	O(n)	Finding the minimum or maximum element in a binary search tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Successor/Predecessor	O(1)	O(log n)	O(n)	Finding the successor or predecessor of an element in a binary search tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Height	O(1)	O(log n)	O(n)	Determining the height of a binary search tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying a binary search tree.	O(n)	O(n)	O(n)	Memory required to store the copied

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
								binary search tree is linear.

8. operations on 2D arrays:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Accessing an Element	O(1)	O(1)	O(1)	Accessing an element in a 2D array requires constant time.	O(1)	O(1)	O(1)	Memory required to store the accessed element is constant.
Insertion at End (Row Append)	O(1)	O(1)	O(1)	Inserting a row at the end of a 2D array requires constant time.	O(1)	O(1)	O(1)	Memory required remains constant.
Insertion at Beginning (Row Prepend)	O(n)	O(n)	O(n)	Inserting a row at the beginning of a 2D array requires shifting elements, which takes linear time.	O(1)	O(n)	O(n)	Memory required remains constant.
Deletion of Row	O(1)	O(n)	O(n)	Deleting a row from a 2D array requires shifting elements, which takes linear time.	O(1)	O(n)	O(n)	Memory required remains constant.
Insertion at End (Column Append)	O(1)	O(m)	O(m)	Inserting a column at the end of a 2D array requires updating each row, which takes linear time.	O(m)	O(m)	O(m)	Memory required increases linearly with the number of columns.
Insertion at Beginning (Column Prepend)	O(n)	O(n * m)	O(n * m)	Inserting a column at the beginning of a 2D array requires shifting elements, which takes linear time.	O(n * m)	O(n * m)	O(n * m)	Memory required increases linearly with the size of the array.
Deletion of Column	O(1)	O(n * m)	O(n * m)	Deleting a column from a 2D array requires updating each row, which takes linear time.	O(n * m)	O(n * m)	O(n * m)	Memory required remains constant.
Searching (Linear Search)	O(n * m)	O(n * m)	O(n * m)	Searching for an element using linear search requires traversing the entire array.	O(1)	O(1)	O(1)	Memory required remains constant.
Searching (Binary Search)	O(n * log m)	O(n * log m)	O(n * log m)	Searching for an element using binary search requires a sorted array, so preprocessing may be needed.	O(1)	O(1)	O(1)	Memory required remains constant.
Sorting (Row-wise)	O(n * m * log m)	O(n * m * log m)	O(n * m * log m)	Sorting each row of a 2D array using an efficient sorting algorithm like Quick Sort or Merge Sort.	O(1)	O(1)	O(1)	Memory required remains constant.
Sorting (Column-wise)	O(n * log n * m)	O(n * log n * m)	O(n * log n * m)	Sorting each column of a 2D array using an efficient sorting algorithm like Quick Sort or Merge Sort.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n * m)	O(n * m)	O(n * m)	Copying a 2D array requires iterating through all	O(n * m)	O(n * m)	O(n * m)	Memory required to store the copied

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
				elements.				array is linear.

9. operations on a hash set:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Insertion	O(1)	O(1)	O(n)	Inserting an element into a hash set.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Deletion	O(1)	O(1)	O(n)	Deleting an element from a hash set.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Search	O(1)	O(1)	O(n)	Searching for an element in a hash set.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Copying	O(n)	O(n)	O(n)	Copying a hash set.	O(n)	O(n)	O(n)	Memory required to store the copied hash set is linear.

10. operations on a hash map (hash table):

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Insertion	O(1)	O(1)	O(n)	Inserting a key-value pair into a hash map.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Deletion	O(1)	O(1)	O(n)	Deleting a key-value pair from a hash map.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Search	O(1)	O(1)	O(n)	Searching for a key in a hash map.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Accessing an Element	O(1)	O(1)	O(n)	Accessing a value by key in a hash map.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Copying	O(n)	O(n)	O(n)	Copying a hash map.	O(n)	O(n)	O(n)	Memory required to store the copied hash map is linear.

9. operations on a trie:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Insertion	O(1)	O(m)	O(m)	Inserting a word into a trie, where m is the length of the word.	O(1)	O(m)	O(m)	Memory required increases linearly with the length of the word.

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Deletion	O(1)	O(m)	O(m)	Deleting a word from a trie, where m is the length of the word.	O(1)	O(m)	O(m)	Memory required remains constant, but can increase due to resizing.
Search	O(1)	O(m)	O(m)	Searching for a word in a trie, where m is the length of the word.	O(1)	O(1)	O(1)	Memory required remains constant.
Prefix Search	O(1)	O(k)	O(k)	Finding all words with a given prefix, where k is the length of the prefix.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying a trie.	O(n)	O(n)	O(n)	Memory required to store the copied trie is linear.

10. operations on a hash table:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Insertion	O(1)	O(1)	O(n)	Inserting an element into a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Deletion	O(1)	O(1)	O(n)	Deleting an element from a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Search	O(1)	O(1)	O(n)	Searching for an element in a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Accessing an Element	O(1)	O(1)	O(n)	Accessing a value by key in a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Copying	O(n)	O(n)	O(n)	Copying a hash table.	O(n)	O(n)	O(n)	Memory required to store the copied hash table is linear.

11. operations on an ArrayList:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Accessing an Element	O(1)	O(1)	O(1)	Accessing an element in an ArrayList by index.	O(1)	O(1)	O(1)	Memory required to store the accessed element is constant.
Insertion at End (Append)	O(1)	O(1)	O(n)	Inserting an element at the end of an ArrayList.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to resizing.
Insertion at Beginning	O(n)	O(n)	O(n)	Inserting an element at the beginning of	O(1)	O(n)	O(n)	Memory required remains constant, but

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
				an ArrayList.				can increase due to resizing.
Insertion in Middle	O(n)	O(n)	O(n)	Inserting an element in the middle of an ArrayList.	O(1)	O(n)	O(n)	Memory required remains constant, but can increase due to resizing.
Deletion at End	O(1)	O(1)	O(1)	Deleting an element from the end of an ArrayList.	O(1)	O(1)	O(1)	Memory required remains constant.
Deletion at Beginning	O(n)	O(n)	O(n)	Deleting an element from the beginning of an ArrayList.	O(1)	O(n)	O(n)	Memory required remains constant, but can increase due to resizing.
Deletion in Middle	O(n)	O(n)	O(n)	Deleting an element from the middle of an ArrayList.	O(1)	O(n)	O(n)	Memory required remains constant, but can increase due to resizing.
Copying	O(n)	O(n)	O(n)	Copying an ArrayList.	O(n)	O(n)	O(n)	Memory required to store the copied ArrayList is linear.

12. divide and conquer algorithm:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Divide	O(1)	O(1)	O(1)	Dividing the problem into subproblems.	O(1)	O(1)	O(1)	Memory required remains constant.
Conquer	O(1)	O(n log n)	O(n^2)	Solving each subproblem recursively.	O(1)	O(1)	O(1)	Memory required remains constant.
Merge (if applicable)	O(n)	O(n)	O(n)	Merging the solutions of subproblems.	O(n)	O(n)	O(n)	Memory required linearly depends on the size of the solution.

13. complexities for hashing:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Insertion	O(1)	O(1)	O(n)	Inserting an element into a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to collisions or resizing.
Deletion	O(1)	O(1)	O(n)	Deleting an element from a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to collisions or resizing.
Search	O(1)	O(1)	O(n)	Searching for an element in a hash table.	O(1)	O(1)	O(n)	Memory required remains constant, but can increase due to collisions or resizing.

13. generic greedy algorithm:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Greedy Algorithm	O(n)	O(n)	O(n)	Solving the problem using a greedy strategy.	O(1)	O(1)	O(1)	Memory required remains constant.

14. generic backtracking algorithm:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Backtracking Algorithm	O(1)	O(b^d)	O(b^d)	Solving the problem using backtracking, where b is the branching factor and d is the depth of the search tree.	O(d)	O(d)	O(d)	Memory required increases with the depth of the recursive calls.

15. tree traversal algorithms:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Depth-First Traversal	O(n)	O(n)	O(n)	Visiting each node once using a depth-first traversal algorithm.	O(h)	O(h)	O(h)	Memory required is proportional to the height of the tree.
Breadth-First Traversal	O(n)	O(n)	O(n)	Visiting each node once using a breadth-first traversal algorithm.	O(w)	O(w)	O(w)	Memory required is proportional to the maximum width of the tree (w).

17.dynamic programming algorithms:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Dynamic Programming	O(n)	O(n * m)	O(n * m)	Solving the problem using dynamic programming, where n and m represent the size of input data or dimensions of the problem space.	O(n * m)	O(n * m)	O(n * m)	Memory required is proportional to the size of the problem space or dimensions of the problem.

16. AVL tree:

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Search	O(log n)	O(log n)	O(log n)	Searching for an element in an AVL tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Insertion	O(log n)	O(log n)	O(log n)	Inserting an element into an AVL tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Deletion	O(log n)	O(log n)	O(log n)	Deleting an element from an AVL tree.	O(1)	O(1)	O(1)	Memory required remains constant.

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Traversal (Inorder, Preorder, Postorder)	O(n)	O(n)	O(n)	Traversing all nodes of an AVL tree.	O(n)	O(n)	O(n)	Memory required to store the traversal result is linear.
Minimum/Maximum	O(log n)	O(log n)	O(log n)	Finding the minimum or maximum element in an AVL tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Successor/Predecessor	O(log n)	O(log n)	O(log n)	Finding the successor or predecessor of an element in an AVL tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Height	O(1)	O(1)	O(1)	Determining the height of an AVL tree.	O(1)	O(1)	O(1)	Memory required remains constant.
Copying	O(n)	O(n)	O(n)	Copying an AVL tree.	O(n)	O(n)	O(n)	Memory required to store the copied AVL tree is linear.

17. sorting techniques

Sorting Technique	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Bubble Sort	O(n)	O(n^2)	O(n^2)	Simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	O(1)	O(1)	O(1)	Memory required remains constant.
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Sorting algorithm that divides the input list into two parts: sorted and unsorted. Finds the smallest element from the unsorted part and moves it to the sorted part.	O(1)	O(1)	O(1)	Memory required remains constant.
Insertion Sort	O(n)	O(n^2)	O(n^2)	Sorting algorithm that builds a sorted array one element at a time by repeatedly taking the next element and inserting it into its correct position in the sorted array.	O(1)	O(1)	O(1)	Memory required remains constant.
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Divide and conquer sorting algorithm that recursively divides the input array into halves, sorts them independently, and then merges the sorted halves.	O(n)	O(n)	O(n)	Memory required linearly increases with the size of the input array.
Quick Sort	O(n log n)	O(n log n)	O(n^2)	Divide and conquer sorting algorithm that selects a pivot element and partitions the array into two sub-arrays around the	O(log n)	O(log n)	O(n)	Memory required depends on the recursion depth and pivot selection.

Sorting Technique	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
				pivot. Recursively sorts the sub-arrays.				
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Comparison-based sorting algorithm that uses a binary heap data structure to build a sorted array.	O(1)	O(1)	O(1)	Memory required remains constant.
Counting Sort	O(n + k)	O(n + k)	O(n + k)	Integer sorting algorithm that counts the number of occurrences of each distinct element and then performs prefix sum to determine the position of each element in the output array.	O(n + k)	O(n + k)	O(n + k)	Memory required linearly depends on the range of input elements (k).
Radix Sort	O(n * k)	O(n * k)	O(n * k)	Non-comparison based sorting algorithm that sorts numbers by processing individual digits or groups of digits from least significant to most significant.	O(n + k)	O(n + k)	O(n + k)	Memory required linearly depends on the range of input elements (k).
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	Distribution-based sorting algorithm that distributes elements into a finite number of buckets, sorts each bucket individually, and then concatenates them.	O(n)	O(n + k)	O(n^2)	Memory required depends on the number of buckets and range of input elements (k).
Cyclic Sort	O(n)	O(n^2)	O(n^2)	Sorting algorithm that iterates through the array and places each element in its correct position by swapping elements until the array is sorted.	O(1)	O(1)	O(1)	Memory required remains constant.

18. HEAP

Operation	Best Case Time	Average Case Time	Worst Case Time	Time Complexity Explanation	Best Case Space	Average Case Space	Worst Case Space	Space Complexity Explanation
Insertion	O(log n)	O(log n)	O(log n)	Inserting an element into heap	O(1)	O(1)	O(1)	Memory required remains constant
Deletion (extractMin)	O(log n)	O(log n)	O(log n)	Removing minimum element	O(1)	O(1)	O(1)	Memory required remains constant
Decrease Key	O(log n)	O(log n)	O(log n)	Decreasing key value	O(1)	O(1)	O(1)	Memory required remains constant
Delete	O(log n)	O(log n)	O(log n)	Deleting an element	O(1)	O(1)	O(1)	Memory required remains constant
Heapify (buildHeap)	O(n)	O(n)	O(n)	Building a heap from an array	O(1)	O(1)	O(1)	Memory required remains constant
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Sorting using heap sort	O(1)	O(1)	O(1)	Memory required remains constant