Complexity

Algorithm:

An algorithm is a step-by-step set of instructions or a sequence of operations designed to perform a specific task or solve a particular problem.

Types of Algorithms:

1. **Sorting Algorithms:**
    - Examples: Bubble Sort, Quick Sort, Merge Sort.
    - Purpose: Rearrange elements in a specific order (e.g., ascending or descending).
2. **Searching Algorithms:**
    - Examples: Linear Search, Binary Search.
    - Purpose: Locate a specific element in a collection of data.
3. **Graph Algorithms:**
    - Examples: Depth-First Search (DFS), Breadth-First Search (BFS).
    - Purpose: Analyze relationships between entities represented as vertices and edges.
4. **Greedy Algorithms:**

- Examples: Dijkstra's algorithm for finding the shortest path.
- Purpose: Make locally optimal choices at each stage to achieve a global optimum.

5. **Divide and Conquer Algorithms:**
   - Examples: Merge Sort, Quick Sort.
   - Purpose: Break a problem into smaller subproblems, solve them, and then combine the solutions.
6. **Recursive Algorithms:**
   - Examples: Factorial calculation, Tower of Hanoi.
   - Purpose: Solve a problem by solving smaller instances of the same problem.

Properties of Algorithms:

1. **Input and Output:**
   - **Input:** Algorithms take input data or parameters.
   - **Output:** Algorithms produce output, which is the result or solution.
2. **Definiteness:**
   - Every step of the algorithm must be precisely and unambiguously defined.
3. **Finiteness:**
   - Algorithms must terminate after a finite number of steps.
4. **Effectiveness:**
   - Every operation in the algorithm must be feasible and should lead to the desired result.

5. **Uniqueness:**
   - For a given input, an algorithm should produce a unique output.
6. **Robustness:**
   - An algorithm should be able to handle different inputs, including unexpected or erroneous data, without crashing or producing incorrect results. It should be robust against variations in the input.

Complexity:

The study of complexity helps in understanding how the performance of algorithms or problems scales with increasing input sizes. There are two main types of complexity: time complexity and space complexity.

1. **Time Complexity:**
   - **Definition:** Time complexity measures the amount of time an algorithm takes to complete as a function of the size of the input.
   - **Notation:** Time complexity is often expressed using big-O notation (e.g., O(n), O(n log n), O(n^2)).
   - **Types:**
     - **Best-case time complexity:** The minimum time an algorithm requires for a given input.
     - **Average-case time complexity:** The average time an algorithm requires for all possible inputs.
     - **Worst-case time complexity:** The maximum time an algorithm requires for a given input.
2. **Space Complexity:**
   - **Definition:** Space complexity measures the amount of memory or storage space an algorithm requires as a function of the size of the input.

- **Notation:** Space complexity is also expressed using big-O notation.
- **Types:**
  - **Auxiliary space complexity:** The extra space needed by the algorithm, excluding the input space.
  - **Total space complexity:** The total space, including both input and auxiliary space.

Asymptotic Time Complexity:

Asymptotic time complexity is a measure of the computational efficiency of an algorithm, expressing how its running time grows relative to the size of the input in the worst-case scenario.

Asymptotic Notation:

1. Big-oh Notation(O)

   $f(n)=O(g(n)) \approx f(n) \leq g(n)$

2. Big-omega Notation(Ω)

   $f(n)=\Omega(g(n)) \approx f(n) \geq g(n)$

3. Theta Notation(θ)

   $f(n)=\theta(g(n)) \approx f(n) = g(n)$

4. Little-oh Notation(o)

   $f(n)=o(g(n)) \approx f(n) < g(n)$

5. Little-omega Notation(ω)

$f(n)=\omega(g(n)) \approx f(n) > g(n)$

Types of Time Complexity:

1. **O(1) - Constant Time Complexity:**
   - The running time of the algorithm remains constant, regardless of the input size.
2. **O(log n) - Logarithmic Time Complexity:**
   - The running time grows logarithmically with the size of the input. Common in algorithms with divide-and-conquer strategies.
3. **O(n) - Linear Time Complexity:**
   - The running time grows linearly with the size of the input. For every additional input element, the running time increases proportionally.
4. **O(n log n) - Linearithmic Time Complexity:**
   - Common in efficient sorting algorithms like Merge Sort and Heap Sort.
5. **O(n^2) - Quadratic Time Complexity:**
   - The running time is proportional to the square of the size of the input. Common in nested loops where each element is compared with every other element.
6. **O(n^k) - Polynomial Time Complexity:**
   - The running time is a polynomial function of the input size, where k is a constant.
7. **O(2^n) - Exponential Time Complexity:**

- The running time doubles with each additional element in the input. Common in algorithms with recursive backtracking.

8. **O(n!) - Factorial Time Complexity:**

   - The running time grows factorially with the size of the input. Common in brute-force algorithms that consider all possible permutations.