Time complexity measures how the performance of an algorithm changes with input size (n). Big-O notation expresses the worst-case complexity. For example, **O(1)** means constant time; **O(n)** means linear time; **O(n²)** represents quadratic time; **O(log n)** is logarithmic; and **O(n log n)** represents efficient divide-and-conquer algorithms like merge sort.

Searching algorithms determine how to locate a specific value in a dataset. The simplest is **linear search (O(n))**, which checks each element sequentially. It works on unsorted data but is slow for large inputs. **Binary search (O(log n))** works on sorted arrays by repeatedly dividing the search space in half, dramatically improving efficiency.

Sorting algorithms arrange elements in ascending or descending order:

- **Bubble Sort:** Repeatedly swaps adjacent elements—simple but slow (O(n²)).
- **Selection Sort:** Selects the smallest element in each iteration—also O(n²).
- **Insertion Sort:** Efficient for small or nearly sorted arrays (O(n²) worst case).
- **Merge Sort:** A divide-and-conquer algorithm that splits, sorts, and merges; stable and fast with O(n log n) time.
- **Quick Sort:** Uses a pivot to partition the array; average O(n log n) but worst O(n²). One of the fastest practical algorithms.
- **Heap Sort:** Builds a heap and extracts the maximum/minimum repeatedly; guarantees O(n log n).

Understanding time complexity helps you choose the right algorithm for large-scale problems. Sorting and searching form the foundation of most real-world applications, from databases to competitive programming.