**1. Quick Sort (Divide and Conquer)**

**Algorithm:**

1. If the array has 0 or 1 elements, it's already sorted.
2. Pick a pivot element (e.g., middle).
3. Partition the array into three parts:
   * Elements less than the pivot.
   * Elements equal to the pivot.
   * Elements greater than the pivot.
4. Recursively apply Quick Sort to the left and right partitions.
5. Combine the sorted parts.

 **Complexity:**

* Best: O(n log n)
* Average: O(n log n)
* Worst: O(n²)
* Space: O(log n)

 **Applications:**

* Sorting in libraries (e.g., Python’s sort)
* Large datasets
* Databases

 **Advantages:**

* Fast in practice (average case)
* In-place sorting

 **Disadvantages:**

* Poor worst-case performance
* Not stable

**2. Selection Sort**

**Algorithm:**

1. Repeat for each index i in the array:
   * Find the smallest element in the unsorted part.
   * Swap it with the element at index i.
2. Repeat until the whole array is sorted.

 **Complexity:**

* Time: O(n²)
* Space: O(1)

 **Applications:**

* Small lists
* When memory write is limited

 **Advantages:**

* Simple to implement
* No extra space

 **Disadvantages:**

* Inefficient on large lists
* Not stable

**3. Insertion Sort**

**Algorithm:**

1. Start from the second element.
2. Compare it with all elements before it.
3. Insert it into its correct position in the sorted part.
4. Repeat for all elements.

 **Complexity:**

* Best: O(n)
* Average/Worst: O(n²)
* Space: O(1)

 **Applications:**

* Small datasets
* Online sorting

 **Advantages:**

* Simple and intuitive
* Efficient for nearly sorted arrays

 **Disadvantages:**

* Poor performance on large lists
* Not suitable for large datasets

**4. Naive String Matching**

**Algorithm:**

1. Loop over each index i in the text.
2. Compare substring of text with the pattern from position i.
3. If match found, return the index.
4. If no match found, return -1.

 **Complexity:**

* Best: O(n)
* Worst: O(n \* m)
* Space: O(1)

 **Applications:**

* Simple pattern matching
* DNA/protein sequence matching (small)

 **Advantages:**

* Easy to implement

 **Disadvantages:**

* Slow for large text/pattern.

**5. N-Queens Problem (Backtracking)**

**Algorithm:**

1. Place a queen in row 0.
2. For each next row:
   * Try all columns.
   * Check if placing a queen is safe.
   * If safe, place and move to next row.
   * If no column works, backtrack.
3. When all rows are filled, store the solution.
4. **Complexity:**
   * Time: O(N!)
   * Space: O(N)
5. **Applications:**
   * Puzzle solving
   * AI constraint satisfaction problems
6. **Advantages:**
   * Finds all valid solutions
7. **Disadvantages:**
   * Inefficient for large N
   * Backtracking can be slow

**6. Longest Common Subsequence (LCS)**

**Algorithm:**

1. Create a 2D DP table dp[m+1][n+1].
2. Fill table:
   * If characters match: dp[i][j] = dp[i-1][j-1] + 1.
   * Else: dp[i][j] = max(dp[i-1][j], dp[i][j-1]).
3. dp[m][n] gives length of LCS.

 **Complexity:**

* Time: O(m \* n)
* Space: O(m \* n)

 **Applications:**

* File comparison
* DNA sequence analysis

 **Advantages:**

* Guarantees correct LCS

 **Disadvantages:**

* High space usage for large strings

**7. Floyd-Warshall Algorithm**

**Algorithm:**

1. Initialize distance matrix from the input graph.
2. For every vertex k:
   * For every pair (i, j):
     + Update dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).
3. The matrix gives shortest paths between all pairs.

 **Complexity:**

* Time: O(V³)
* Space: O(V²)

 **Applications:**

* All-pairs shortest path
* Network routing

 **Advantages:**

* Handles negative weights
* Simple logic

 **Disadvantages:**

* Inefficient for large graphs

**8. Kruskal’s Algorithm**

**Algorithm:**

1. Sort all edges by weight.
2. Initialize an empty MST and disjoint sets.
3. For each edge:
   * If adding it doesn't form a cycle, include it in MST.
4. Stop when MST has (V-1) edges.

 **Complexity:**

* Time: O(E log E)
* Space: O(V)

 **Applications:**

* Network design
* Clustering

 **Advantages:**

* Works well on sparse graphs

 **Disadvantages:**

* Needs disjoint set structure

**9. Dijkstra’s Algorithm**

**Algorithm:**

1. Set all distances to infinity, except start node = 0.
2. Use a priority queue to explore the node with the smallest distance.
3. For each neighbor, update the distance if a shorter path is found.
4. Repeat until all nodes are visited.

 **Complexity:**

* Time: O((V + E) log V) with min-heap
* Space: O(V)

 **Applications:**

* GPS navigation
* Shortest path routing

 **Advantages:**

* Efficient with proper data structures

 **Disadvantages:**

* Doesn’t handle negative weights

**10. Strassen’s Matrix Multiplication**

**Algorithm:**

1. Split matrices A and B into 4 submatrices.
2. Compute 7 matrix products (M1 to M7) using recursive Strassen's calls.
3. Combine the products to form the result matrix.
4. Repeat until size is 1×1 (base case).

 **Complexity:**

* Time: O(n^2.81)
* Space: O(n²)

 **Applications:**

* High-performance computing
* Scientific matrix computation

 **Advantages:**

* Faster than standard algorithm

 **Disadvantages:**

* Complex to implement
* Not practical for small matrices

**11. *15-Puzzle Problem (A Search)*\***

**Algorithm:**

1. Use A\* search with Manhattan distance as the heuristic.
2. Start with initial board in priority queue.
3. In each step:
   * Expand the board with the lowest f = g + h.
   * Generate valid next moves.
   * Add new states to the queue if not visited.
4. Stop when goal state is reached.

 **Complexity:**

* Time: O(N!) worst-case
* Space: O(N²) per node

 **Applications:**

* Game solving
* AI pathfinding

 **Advantages:**

* Finds optimal solution

 **Disadvantages:**

* High memory and time for complex inputs