Here's the continuation of the list of **Essential DSA Algorithms**:

---

1. Arrays:

- **Kadane's Algorithm**: Finds the maximum sum of a contiguous subarray.
    - *Use Case*: Identifying the largest sum subarray in problems like Leetcode #53.
    - *Resource*: Kadane's Algorithm Explained
- **Dutch National Flag Algorithm**: Sorts an array containing 0s, 1s, and 2s efficiently.
    - *Use Case*: Sorting arrays with three distinct elements, as in Leetcode #75.
    - *Resource*: Dutch National Flag Problem
- **Two Pointer Technique**: Solves pair and triplet sum problems by using two pointers to traverse the array.
    - *Use Case*: Finding pairs with a given sum in a sorted array, such as Leetcode #167.
    - *Resource*: Two Pointer Technique
- **Sliding Window Technique**: Handles subarray problems with fixed or variable window sizes to optimize performance.
    - *Use Case*: Finding the maximum sum of K consecutive elements, like in Leetcode #209.
    - *Resource*: Sliding Window Technique
- **Prefix Sum and Difference Array**: Allows for fast range sum queries by preprocessing the array.
    - *Use Case*: Calculating the sum of all subarrays efficiently, as seen in Leetcode #303.
    - *Resource*: Prefix Sum Array
- **Moore's Voting Algorithm**: Identifies the majority element in an array that appears more than N/2 times.
    - *Use Case*: Finding the majority element in an array, such as Leetcode #169.
    - *Resource*: Moore's Voting Algorithm
- **Merge Intervals**: Merges overlapping intervals in a list.
    - *Use Case*: Combining overlapping intervals, as in Leetcode #56.

    - *Resource*: Merge Overlapping Intervals
- **Spiral Matrix Traversal**: Traverses a 2D matrix in a spiral order.
    - *Use Case*: Printing a matrix in spiral order, like in Leetcode #54.
    - *Resource*: Spiral Matrix Traversal

---

2. Strings:

- **Rabin-Karp Algorithm**: Performs fast substring search using hashing.
    - *Use Case*: Finding if a given pattern exists in a string, as in Leetcode #28.
    - *Resource*: Rabin-Karp Algorithm
- **KMP Algorithm**: Efficiently matches patterns within strings.
    - *Use Case*: Finding all occurrences of a pattern in a string, such as Leetcode #214.
    - *Resource*: KMP Algorithm
- **Z-Algorithm**: Performs fast pattern searching in linear time.
    - *Use Case*: Finding occurrences of a pattern in a string.
    - *Resource*: Z-Algorithm
- **Manacher's Algorithm**: Finds the longest palindromic substring in linear time.
    - *Use Case*: Identifying the longest palindromic substring, as in Leetcode #5.
    - *Resource*: Manacher's Algorithm
- **Longest Common Subsequence (LCS)**: Determines the longest subsequence common to two sequences.
    - *Use Case*: Finding the longest subsequence common in two strings, like in Leetcode #1143.
    - *Resource*: Longest Common Subsequence
- **Trie (Prefix Tree)**: Facilitates efficient prefix-based searching.
    - *Use Case*: Implementing auto-complete functionality, as in Leetcode #208.
    - *Resource*: [Trie Data Structure](https://www

---

3. Linked Lists:

- **Reverse a Linked List**: Reverses a singly linked list iteratively or recursively.

    - *Use Case*: Reversing linked lists, as in Leetcode #206.
    - *Resource*: Reverse a Linked List
- **Detect and Remove Loop in a Linked List (Floyd's Cycle Detection Algorithm)**: Detects cycles in a linked list using the slow and fast pointer approach.
    - *Use Case*: Detecting cycles in linked lists, as in Leetcode #141.
    - *Resource*: Floyd's Cycle Detection
- **Merge Two Sorted Linked Lists**: Merges two sorted linked lists into one sorted list.
    - *Use Case*: Merging sorted linked lists, as in Leetcode #21.
    - *Resource*: Merge Sorted Linked Lists
- **Find Intersection of Two Linked Lists**: Determines the intersection point of two linked lists.
    - *Use Case*: Finding intersection nodes, as in Leetcode #160.
    - *Resource*: Intersection of Linked Lists
- **Clone a Linked List with Random Pointers**: Copies a linked list with additional random pointers.
    - *Use Case*: Copying linked lists with random pointers, as in Leetcode #138.
    - *Resource*: Clone a Linked List

---

4. Stacks & Queues:

- **Implement Stack using Queues**: Implements a stack using two queues.
    - *Use Case*: Stack implementation, as in Leetcode #225.
    - *Resource*: Stack using Queues
- **Implement Queue using Stacks**: Implements a queue using two stacks.
    - *Use Case*: Queue implementation, as in Leetcode #232.
    - *Resource*: Queue using Stacks
- **Balanced Parentheses (Valid Parentheses Check)**: Checks if parentheses in an expression are balanced.
    - *Use Case*: Validating expressions, as in Leetcode #20.
    - *Resource*: Balanced Parentheses
- **Next Greater Element**: Finds the next greater element for each element in an array.
    - *Use Case*: Finding next greater elements, as in Leetcode #496.

    - *Resource*: Next Greater Element
- **LRU Cache Implementation**: Implements a Least Recently Used (LRU) cache using a hashmap and a doubly linked list.
    - *Use Case*: Caching, as in Leetcode #146.
    - *Resource*: LRU Cache Implementation

---

5. Recursion & Backtracking:

- **N-Queens Problem**: Places N queens on an NxN chessboard so that no two queens attack each other.
    - *Use Case*: Solving N-Queens, as in Leetcode #51.
    - *Resource*: N-Queens Problem
- **Sudoku Solver**: Fills an incomplete Sudoku board using backtracking.
    - *Use Case*: Solving Sudoku puzzles, as in Leetcode #37.
    - *Resource*: Sudoku Solver
- **Word Search in a Matrix**: Finds words in a matrix using DFS and backtracking.
    - *Use Case*: Searching words in matrices, as in Leetcode #79.
    - *Resource*: Word Search

---

6. Trees & Graphs:

- **Binary Tree Inorder, Preorder, and Postorder Traversal**
    - *Use Case*: Tree traversal, as in Leetcode #94.
    - *Resource*: Tree Traversals
- **Level Order Traversal (BFS in Trees)**
    - *Use Case*: Traversing trees level by level, as in Leetcode #102.
    - *Resource*: Level Order Traversal
- **Lowest Common Ancestor (LCA) in a Binary Tree**
    - *Use Case*: Finding LCA, as in Leetcode #236.
    - *Resource*: LCA in Binary Tree
- **Dijkstra's Algorithm (Shortest Path in Graphs)**
    - *Use Case*: Shortest path problems, as in Leetcode #743.

    - *Resource*: Dijkstra's Algorithm
- **Floyd-Warshall Algorithm (All Pairs Shortest Path)**
    - *Use Case*: Finding shortest paths in weighted graphs.
    - *Resource*: Floyd-Warshall Algorithm
- **Kruskal's Algorithm (Minimum Spanning Tree - MST)**
    - *Use Case*: Constructing minimum spanning trees.
    - *Resource*: Kruskal's Algorithm

---

7. Dynamic Programming (DP):

- **0/1 Knapsack Problem**: Selects items to maximize profit while staying within weight constraints.
    - *Use Case*: Knapsack problems, as in Leetcode #416.
    - *Resource*: 0/1 Knapsack Problem
- **Longest Increasing Subsequence (LIS)**
    - *Use Case*: Finding LIS, as in Leetcode #300.
    - *Resource*: LIS
- **Edit Distance (Levenshtein Distance)**
    - *Use Case*: String transformation problems, as in Leetcode #72.
    - *Resource*: Edit Distance

This guide covers foundational **DSA algorithms** essential for **coding interviews** and **competitive programming**.