**1. Reverse Linked List**

**Basic Concept:**  
Reverse the direction of a singly linked list, so that the last node becomes the head, and every node points to its previous node.

**Best Approach:**  
Iterative reversal using three pointers (prev, current, next).

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**2. Merge Two Sorted Lists**

**Basic Concept:**  
Merge two singly linked lists sorted in ascending order into a single sorted list.

**Best Approach:**  
Iterate through both lists, compare node values, and build the merged list by always choosing the smaller current node.

**Time Complexity:** O(m + n)  
**Space Complexity:** O(1) (if reusing nodes; else O(m + n) if building a new list)

**3. Palindrome Linked List**

**Basic Concept:**  
Check if the linked list is the same forward and backward (i.e., forms a palindrome).

**Best Approach:**  
Find the middle, reverse the second half in-place, compare both halves, then (optionally) restore the list.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**4. Odd Even Linked List**

**Basic Concept:**  
Rearrange the list so all nodes at odd indices appear before those at even indices, keeping original order within each group.

**Best Approach:**  
Traverse list with separate odd and even pointers and reconnect lists at the end.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**5. Swapping Nodes in a Linked List (in Pairs)**

**Basic Concept:**  
Swap every two adjacent nodes, e.g., 1-2-3-4 → 2-1-4-3

**Best Approach:**  
Iterate with a dummy node pointer, and swap pointers every two nodes.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**6. Delete the Middle Node of a Linked List**

**Basic Concept:**  
Remove the node at the middle position (for even-length lists, typically remove the second middle node).

**Best Approach:**  
Use fast and slow pointers to find the middle, then remove it.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**7. Remove Nth Node from End of List**

**Basic Concept:**  
Remove the nth node from the end (e.g., last node for n=1).

**Best Approach:**  
Use two pointers: start both at head with a gap of n; when the front reaches end, the other is at the target node.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**8. Remove Duplicates from Sorted List**

**Basic Concept:**  
Remove consecutive duplicate nodes so only unique values remain.

**Best Approach:**  
Traverse, and when a node equals the next, bypass next node.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**9. Insert Into Nth Position in List**

**Basic Concept:**  
Insert a new node at the nth position.

**Best Approach:**  
Traverse up to (n-1)th node and re-link pointers for the new node.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**10. Display/List Creation/Count Nodes/Find Kth Element**

**Basic Concept:**  
Basic utility operations: display list, create by appending, count nodes, return kth node.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**11. Rotate List**

**Basic Concept:**  
Rotate the list to the right by k places (last k nodes come to front).

**Best Approach:**  
Make the list circular, break at the new head.

**Time Complexity:** O(n)  
**Space Complexity:** O(1)

**Summary Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program | Main Concept | Best Time | Best Space | Best Approach |
| Reverse Linked List | In-place reversal | O(n) | O(1) | Three pointers |
| Merge Two Sorted Lists | Merge while traversing | O(m + n) | O(1) | Pointer comparisons |
| Palindrome Linked List | Is list a palindrome | O(n) | O(1) | Half-reverse, compare |
| Odd Even Linked List | Odd indices before even | O(n) | O(1) | Split pointers |
| Swap Nodes in Pairs | Swap every two adjacents | O(n) | O(1) | Pointer swaps with dummy |
| Delete Middle Node | Remove middle | O(n) | O(1) | Fast & slow pointers |
| Remove Nth Node from End | Remove nth from end | O(n) | O(1) | Two-pointer gap |
| Remove Duplicates from Sorted List | Only unique nodes remain | O(n) | O(1) | Skip duplicates |
| Insert into Nth Position | Insert at index | O(n) | O(1) | Traverse and relink |
| Display/Count/Find Kth | Basic traversal/utility | O(n) | O(1) | Simple iteration |
| Rotate List | Rotate right by k | O(n) | O(1) | Circle & break at new head |

**1. Block Swap Algorithm (for Array Rotation)**

**Concept:**  
Rotate an array left by d positions efficiently, in-place and without extra memory. Divide the array into two blocks (A: first d elements, B: rest), and swap blocks (or recursively swap parts) until the rotation is complete.

**Efficient Approach:**

* Recursively swap blocks/parts of blocks as per their lengths.
* Finish when both blocks are equal in size, or keep swapping/reducing until so.

**Best Time Complexity:**

* **O(n)** (every element is swapped at most once).

**Best Space Complexity:**

* **O(1)** (no extra array, in-place swaps).

**Java Key Points:**

* Use iterative or recursive implementation.
* Swap with a helper method that swaps subsets in-place.
* Minimal auxiliary storage.

**2. Maximum Product Subarray**

**Concept:**  
Find the contiguous subarray within an array that has the largest product. Because of possible negative numbers and zeros, simply keeping one running product doesn't work.

**Efficient Approach (Prefix-Suffix/Kadane’s-like):**

* Traverse the array and keep track of both current maximum and minimum products (since negative × negative = positive).
* At every step, update max and min product (especially important if the current number is negative).
* Reset the running product to 1 when zero is encountered to handle subarrays breaking on zero.

**Best Time Complexity:**

* **O(n)** (single pass, updating local min/max at each step).

**Best Space Complexity:**

* **O(1)** (only variables for products and answer).

**Java Key Points:**

* Variables for max\_product, min\_product, and result.
* Single loop to iterate array.

**3. Maximum Sum of Hourglass in Matrix**

**Concept:**  
In a 2D matrix, define an "hourglass" shape (typically 3x3: top three, middle one, and bottom three elements) and find the maximum sum over all possible hourglasses in the matrix.

**Efficient Approach:**

* Loop over every possible hourglass center (not on the edge).
* For each, calculate the sum of its 7 elements with direct indexing.
* Track the largest sum found.

**Best Time Complexity:**

* **O(m\*n)** (m = number of rows, n = columns; every non-border cell is a center).

**Best Space Complexity:**

* **O(1)** (only variables to store current and max sum).

**Java Key Points:**

* Nested loops for all valid centers.
* Simple sum calculation in each.

**Summary Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program | Main Concept | Best Time | Best Space | Best Approach in Java |
| Block Swap Algorithm | Rotate array in-place by d | O(n) | O(1) | In-place recursive/iterative swap |
| Max Product Subarray | Max contiguous product in array | O(n) | O(1) | Track max/min in one pass |
| Maximum Hourglass Sum in Matrix | Max 3x3 "hourglass" sum in 2D array | O(m\*n) | O(1) | Double loop, sum fixed indices |