LINKED LIST

- 2
- 19
- 21
- 23
- 24
- 25
- 61
- 82
- 83
- 86
- 92
- 138
- 141
- 142
- 143
- 146
- 147
- 148
- 160
- 203
- 206
- 234
- 328
- 355
- 445
- 622
- 707
- 817
- 876
- 1019
- 1171
- 1290
- 1669
- 1670
- 1721
- 2058
- 2074
- 2095
- 2130
- 2181
- 2289
- 2487
- 2807
- 2816
- 3217

2 : Problem: **Add Two Numbers**

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:
Input: `l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: `342 + 465 = 807` (since the digits are stored in reverse order).

Example 2:
Input: l1 = [0], l2 = [0]
Output: [0]
Explanation: 0 + 0 = 0.

Example 3:
Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]
Explanation: 9999999 + 9999 = 10009998.

Solution 1: Iterative Approach (Recommended)

Time Complexity: O(max(m,n))
Space Complexity: O(max(m,n))

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;
        int carry = 0;

        while (l1 || l2 || carry) {
            int sum = carry;
            if (l1) {
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2) {
                sum += l2->val;
                l2 = l2->next;
            }
            carry = sum / 10;
            curr->next = new ListNode(sum % 10);
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 2: Recursive Approach

Time Complexity: O(max(m,n))
Space Complexity: O(max(m,n)) - due to recursion stack

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2,
int carry = 0) {
        if (!l1 && !l2 && !carry) return nullptr;

        int sum = carry;
        if (l1) {
            sum += l1->val;
            l1 = l1->next;
        }
        if (l2) {
            sum += l2->val;
            l2 = l2->next;
        }

        ListNode* node = new ListNode(sum % 10);
        node->next = addTwoNumbers(l1, l2, sum / 10);
        return node;
    }
};
```

Solution 3: Convert to Numbers (Limited Use)

Time Complexity: O(max(m,n))
Space Complexity: O(max(m,n))
Note: This approach fails for very large numbers due to integer overflow

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        long long num1 = 0, num2 = 0;
        long long multiplier = 1;

        while (l1) {
            num1 += l1->val * multiplier;
            multiplier *= 10;
            l1 = l1->next;
        }

        multiplier = 1;
        while (l2) {
            num2 += l2->val * multiplier;
            multiplier *= 10;
            l2 = l2->next;
        }

        long long sum = num1 + num2;
```

```
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        if (sum == 0) {
            return new ListNode(0);
        }

        while (sum > 0) {
            curr->next = new ListNode(sum % 10);
            curr = curr->next;
            sum /= 10;
        }

        return dummy->next;
    }
};
```

Solution 4: Using Stacks

Time Complexity: O(max(m,n))
Space Complexity: O(m + n) - for storing stacks

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        stack<int> s1, s2;

        while (l1) {
            s1.push(l1->val);
            l1 = l1->next;
        }
        while (l2) {
            s2.push(l2->val);
            l2 = l2->next;
        }

        ListNode* dummy = nullptr;
        int carry = 0;

        while (!s1.empty() || !s2.empty() || carry) {
            int sum = carry;
            if (!s1.empty()) {
                sum += s1.top();
                s1.pop();
            }
            if (!s2.empty()) {
                sum += s2.top();
                s2.pop();
            }
            carry = sum / 10;
            ListNode* node = new ListNode(sum % 10);
            node->next = dummy;
            dummy = node;
        }

        return dummy;
    }
};
```

Solution 5: In-Place Modification

Time Complexity: O(max(m,n))
Space Complexity: O(1) - modifies one input list

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        ListNode* dummy = new ListNode(0);
        dummy->next = l1;
        ListNode* prev = dummy;
        int carry = 0;

        while (l1 || l2 || carry) {
            int sum = carry;
            if (l1) {
                sum += l1->val;
            }
            if (l2) {
                sum += l2->val;
                l2 = l2->next;
            }

            carry = sum / 10;
            if (l1) {
                l1->val = sum % 10;
                prev = l1;
                l1 = l1->next;
            } else {
                prev->next = new ListNode(sum % 10);
                prev = prev->next;
            }
        }

        return dummy->next;
    }
};
```

19 : Problem : **Remove Nth Node From End of List**

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Example 1:
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]

Example 2:
Input: head = [1], n = 1
Output: []

Example 3:
Input: head = [1,2], n = 1
Output: [1]

Approach 1: Two Pass (Calculate Length First)

- First pass: Calculate length of linked list
- Second pass: Move to (length - n)th node and remove it
- Handle edge case when removing head node

Time Complexity: O(n) where L is length of list
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        if (!head) return nullptr;

        // Calculate length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Find position to remove
        int position = length - n;

        // If removing head
        if (position == 0) {
            ListNode* newHead = head->next;
            delete head;
            return newHead;
        }

        // Move to node before the one to remove
        curr = head;
        for (int i = 0; i < position - 1; i++) {
            curr = curr->next;
        }

        // Remove the node
        ListNode* nodeToDelete = curr->next;
        curr->next = curr->next->next;
        delete nodeToDelete;

        return head;
    }
};
```

Approach 2: One Pass (Two Pointers)

- Use two pointers: fast and slow
- Move fast pointer n steps ahead first
- Then move both pointers together until fast reaches end
- Slow will be at node before the one to remove

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

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        ListNode* fast = dummy;
        ListNode* slow = dummy;

        // Move fast n+1 steps ahead
        for (int i = 0; i <= n; i++) {
            fast = fast->next;
        }

        // Move both until fast reaches end
        while (fast) {
            fast = fast->next;
            slow = slow->next;
        }

        // Remove the node
        ListNode* nodeToDelete = slow->next;
        slow->next = slow->next->next;
        delete nodeToDelete;

        ListNode* result = dummy->next;
        delete dummy;
        return result;
    }
};
```

Approach 3: Recursive

- Use recursion to reach the end of list
- Count backwards from the end
- Remove node when count matches n

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        int pos = 0;
        return removeHelper(head, n, pos);
    }

private:
    ListNode* removeHelper(ListNode* node, int n, int&
pos) {
        if (!node) {
            pos = 0;
            return nullptr;
        }

        node->next = removeHelper(node->next, n, pos);
        pos++;

        if (pos == n) {
            ListNode* nextNode = node->next;
            delete node;
            return nextNode;
        }

        return node;
    }
};
```

Approach 4: Stack Based

- Push all nodes to stack
- Pop n nodes to find the node to remove
- Use stack to get previous node

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

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        stack<ListNode*> st;
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* curr = dummy;

        // Push all nodes to stack
        while (curr) {
            st.push(curr);
            curr = curr->next;
        }

        // Pop n nodes
        for (int i = 0; i < n; i++) {
            st.pop();
        }

        // Get the node before the one to remove
        ListNode* prev = st.top();
        ListNode* nodeToDelete = prev->next;
        prev->next = prev->next->next;
        delete nodeToDelete;

        ListNode* result = dummy->next;
        delete dummy;
        return result;
    }
};
```

Approach 5: Using Array/Vector

- Store all nodes in array/vector
- Directly access the node to remove using index

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

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        vector<ListNode*> nodes;
        ListNode* curr = head;

        // Store all nodes in vector
        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        int length = nodes.size();
        int position = length - n;

        // If removing head
        if (position == 0) {
            ListNode* newHead = head->next;
            delete head;
            return newHead;
        }

        // Remove the node
        ListNode* prev = nodes[position - 1];
        ListNode* nodeToDelete = prev->next;
        prev->next = prev->next->next;
        delete nodeToDelete;

        return head;
    }
};
```

Approach 6: Two Pointers without Dummy Node

- Similar to Approach 2 but without dummy node
- Handle edge cases separately

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

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* fast = head;
        ListNode* slow = head;

        // Move fast n steps ahead
        for (int i = 0; i < n; i++) {
            fast = fast->next;
        }

        // If removing head
        if (!fast) {
            ListNode* newHead = head->next;
            delete head;
            return newHead;
        }

        // Move both until fast reaches last node
        while (fast->next) {
            fast = fast->next;
            slow = slow->next;
        }

        // Remove the node
        ListNode* nodeToDelete = slow->next;
        slow->next = slow->next->next;
        delete nodeToDelete;

        return head;
    }
};
```

Approach 7: Reverse and Remove

- Reverse the linked list
- Remove nth node from beginning
- Reverse back

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

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // Reverse the list
        head = reverseList(head);

        // Remove nth node from beginning
        if (n == 1) {
            ListNode* newHead = head->next;
            delete head;
            return reverseList(newHead);
        }

        ListNode* curr = head;
        for (int i = 1; i < n - 1; i++) {
            curr = curr->next;
        }

        ListNode* nodeToDelete = curr->next;
        curr->next = curr->next->next;
        delete nodeToDelete;

        // Reverse back
        return reverseList(head);
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

---

21 : Problem **: Merge Two Sorted Lists**

Merge two sorted linked lists and return it as a sorted list. The list should be made by splicing together the nodes of the first two lists.

Example 1:
Input: `l1 = [1,2,4], l2 = [1,3,4]`
Output: `[1,1,2,3,4,4]`

Example 2:
Input: `l1 = [], l2 = []`
Output: `[]`

Example 3:
Input: `l1 = [], l2 = [0]`
Output: `[0]`

Solution 1: Iterative Approach (Recommended)

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
```

```
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val <= l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        // Attach remaining nodes
        if (l1) curr->next = l1;
        if (l2) curr->next = l2;

        return dummy->next;
    }
};
```

Solution 2: Recursive Approach

Time Complexity: O(m + n)
Space Complexity: O(m + n) - recursion stack

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;

        if (l1->val <= l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

Solution 3: In-Place Modification

Time Complexity: O(m + n)
Space Complexity: O(1) - modifies input lists

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;

        // Ensure l1 always starts with smaller value
        if (l1->val > l2->val) {
            swap(l1, l2);
        }

        ListNode* head = l1;

        while (l1 && l2) {
            ListNode* temp = nullptr;
            while (l1 && l1->val <= l2->val) {
                temp = l1;
                l1 = l1->next;
            }
            temp->next = l2;
            swap(l1, l2);
        }

        return head;
    }
};
```

Solution 4: Using Priority Queue Approach

Time Complexity: O((m+n)log(m+n))
Space Complexity: O(m + n)

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1 && !l2) return nullptr;
        if (!l1) return l2;
        if (!l2) return l1;

        priority_queue<int, vector<int>, greater<int>> pq;

        while (l1) {
            pq.push(l1->val);
            l1 = l1->next;
        }
        while (l2) {
            pq.push(l2->val);
            l2 = l2->next;
        }

        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (!pq.empty()) {
            curr->next = new ListNode(pq.top());
            pq.pop();
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 5: Two Pointers with Extra Space

Time Complexity: O(m + n)
Space Complexity: O(m + n)

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        vector<int> values;

        while (l1) {
            values.push_back(l1->val);
            l1 = l1->next;
        }
        while (l2) {
            values.push_back(l2->val);
            l2 = l2->next;
        }

        sort(values.begin(), values.end());

        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        for (int val : values) {
            curr->next = new ListNode(val);
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

23: Problem: **Merge k Sorted Lists**

You are given an array of `k` linked-lists lists, each linked-list is sorted in ascending order. Merge all the linked-lists into one sorted linked-list and return it.

Example 1:
Input: `lists = [[1,4,5],[1,3,4],[2,6]]`
Output: `[1,1,2,3,4,4,5,6]`
Explanation: The linked-lists are:
1->4->5,
1->3->4,
2->6
Merging them into one sorted list:
1->1->2->3->4->4->5->6

Example 2:
Input: `lists = []`
Output: `[]`

Example 3:
Input: `lists = [[]]`
Output: `[]`

Solution 1: Divide and Conquer (Merge Sort Approach)

Time Complexity: O(N log k) where N is total nodes, k is number of lists
Space Complexity: O(1) excluding recursion stack

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return nullptr;
        return mergeKListsHelper(lists, 0, lists.size() - 1);
    }
private:
    ListNode* mergeKListsHelper(vector<ListNode*>&
lists, int left, int right) {
        if (left == right) return lists[left];

        int mid = left + (right - left) / 2;
        ListNode* l1 = mergeKListsHelper(lists, left, mid);
        ListNode* l2 = mergeKListsHelper(lists, mid + 1,
right);
        return mergeTwoLists(l1, l2);
    }
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val <= l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        if (l1) curr->next = l1;
        if (l2) curr->next = l2;

        return dummy->next;
    }
};
```

Solution 2: Priority Queue (Min-Heap)

Time Complexity: O(N log k)
Space Complexity: O(k)

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        auto compare = [](ListNode* a, ListNode* b) {
```

```
        return a->val > b->val;
    };
    priority_queue<ListNode*, vector<ListNode*>,
decltype(compare)> pq(compare);

  // Push the first node of each list into the priority queue
    for (ListNode* list : lists) {
        if (list) {
            pq.push(list);
        }
    }
    ListNode* dummy = new ListNode(0);
    ListNode* curr = dummy;

    while (!pq.empty()) {
        ListNode* node = pq.top();
        pq.pop();

        curr->next = node;
        curr = curr->next;

        if (node->next) {
            pq.push(node->next);
        }
    }
    return dummy->next;
  }
};
```

Solution 3: Iterative Merge (Pairwise Merging)

Time Complexity: O(N log k)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return nullptr;

        int k = lists.size();
        while (k > 1) {
            for (int i = 0; i < k / 2; i++) {
                lists[i] = mergeTwoLists(lists[i], lists[k - 1 - i]);
            }
            k = (k + 1) / 2;
        }

        return lists[0];
    }

private:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val <= l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        if (l1) curr->next = l1;
        if (l2) curr->next = l2;

        return dummy->next;
    }
};
```

Solution 4: Brute Force (Collect All Nodes)

Time Complexity: O(N log N) where N is total nodes
Space Complexity: O(N)

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        vector<int> values;

        // Collect all values
        for (ListNode* list : lists) {
            while (list) {
                values.push_back(list->val);
                list = list->next;
            }
        }
        // Sort all values
        sort(values.begin(), values.end());

        // Create new linked list
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        for (int val : values) {
            curr->next = new ListNode(val);
            curr = curr->next;
        }
        return dummy->next;
    }
};
```

Solution 5: Sequential Merging

Time Complexity: O(kN) where k is number of lists
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return nullptr;

        ListNode* result = lists[0];
        for (int i = 1; i < lists.size(); i++) {
            result = mergeTwoLists(result, lists[i]);
        }

        return result;
    }

private:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val <= l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        if (l1) curr->next = l1;
        if (l2) curr->next = l2;

        return dummy->next;
    }
};
```

Solution 6: Using Multiset

Time Complexity: O(N log N)
Space Complexity: O(N)

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        multiset<int> values;

        for (ListNode* list : lists) {
            while (list) {
                values.insert(list->val);
                list = list->next;
            }
        }

        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        for (int val : values) {
            curr->next = new ListNode(val);
            curr = curr->next;
        }
        return dummy->next;
    }
};
```

Solution 7: Optimized Priority Queue with Custom Comparator

Time Complexity: O(N log k)
Space Complexity: O(k)

```
struct Compare {
    bool operator()(const ListNode* a, const ListNode* b) {
        return a->val > b->val;
    }
};

class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>,
Compare> pq;

        // Add first nodes of all lists
        for (ListNode* node : lists) {
            if (node) {
                pq.push(node);
            }
        }

        if (pq.empty()) return nullptr;

        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;

        while (!pq.empty()) {
            ListNode* current = pq.top();
            pq.pop();

            tail->next = current;
            tail = tail->next;

            if (current->next) {
                pq.push(current->next);
            }
        }

        return dummy->next;
    }
};
```

24: problem (Swap Nodes in Pairs)

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list nodes (only nodes themselves may be changed).

Example:
Input: 1->2->3->4
Output: 2->1->4->3

Solution 1: Iterative approach

- Use dummy node to handle head swaps
- Maintain prev, current and next pointers
- Swap pairs by adjusting pointers
- Move pointers forward after each swap

Time complexity : O(n)
Space complexity : O(1)

```
ListNode* swapPairs(ListNode* head) {
    ListNode dummy(0);
    dummy.next = head;
    ListNode* prev = &dummy;
    while (prev->next && prev->next->next) {
        ListNode* first = prev->next;
        ListNode* second = first->next;
        first->next = second->next;
        second->next = first;
        prev->next = second;
        prev = first;
    }
    return dummy.next;
}
```

Solution 2 : Recursive approach

- Base case: empty list or single node
- Swap first two nodes recursively
- Connect swapped pair to rest of processed list

Time complexity : O(n)
Space complexity : O(n) due to recursion stack

```
ListNode* swapPairs(ListNode* head) {
    if (!head || !head->next) return head;
    ListNode* newHead = head->next;
    head->next = swapPairs(newHead->next);
    newHead->next = head;
    return newHead;
}
```

Edge cases:
- Empty list
- Single node list
- Odd length list
- Even length list

25 : Problem : Reverse Nodes in k-Group

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:
Input: `head = [1,2,3,4,5], k = 2`
Output: `[2,1,4,3,5]`

Example 2:
Input: `head = [1,2,3,4,5], k = 3`
Output: `[3,2,1,4,5]`

Example 3:
Input: `head = [1,2,3,4,5], k = 1`
Output: `[1,2,3,4,5]`

Example 4:
Input: `head = [1], k = 1`
Output: `[1]`

Solution 1: Iterative Approach (Recommended)

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

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (!head || k == 1) return head;

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;
        ListNode* curr = head;

        int count = 0;
        while (curr) {
            count++;
            curr = curr->next;
        }

        while (count >= k) {
            curr = prev->next;
            ListNode* next = curr->next;

            for (int i = 1; i < k; i++) {
                curr->next = next->next;
                next->next = prev->next;
                prev->next = next;
                next = curr->next;
            }
            prev = curr;
            count -= k;
        }
        return dummy->next;
    }
};
```

Solution 2: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n/k) - recursion stack

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        // Check if there are at least k nodes
        ListNode* curr = head;
        int count = 0;
        while (curr && count < k) {
            curr = curr->next;
            count++;
        }

        // If we have k nodes, reverse them
        if (count == k) {
            // Reverse first k nodes
            ListNode* reversedHead =
reverseLinkedList(head, k);

            // Recursively reverse remaining nodes
            head->next = reverseKGroup(curr, k);

            return reversedHead;
        }

        // Return head if less than k nodes
        return head;
    }

private:
    ListNode* reverseLinkedList(ListNode* head, int k) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        for (int i = 0; i < k; i++) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 3: Stack-Based Approach

Time Complexity: O(n)
Space Complexity: O(k)

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (!head || k == 1) return head;

        stack<ListNode*> st;
        ListNode* dummy = new ListNode(0);
        ListNode* curr = head;
        ListNode* tail = dummy;

        while (curr) {
            // Push k nodes to stack
            int count = 0;
            ListNode* temp = curr;
            while (temp && count < k) {
                st.push(temp);
                temp = temp->next;
                count++;
            }

            // If we have k nodes, reverse them
            if (count == k) {
                while (!st.empty()) {
                    tail->next = st.top();
                    st.pop();
                    tail = tail->next;
                }
                tail->next = temp; // Connect to next segment
                curr = temp;
            } else {
                // Connect remaining nodes as they are
                tail->next = curr;
                break;
            }
        }

        return dummy->next;
    }
};
```

Solution 4: Two-Pass with Count

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

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (!head || k == 1) return head;

        // First pass: count total nodes
        int totalNodes = 0;
        ListNode* temp = head;
```

```
        while (temp) {
            totalNodes++;
            temp = temp->next;
        }

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;
        ListNode* curr = head;

        // Second pass: reverse in groups
        for (int i = 0; i < totalNodes / k; i++) {
            for (int j = 1; j < k; j++) {
                ListNode* next = curr->next;
                curr->next = next->next;
                next->next = prev->next;
                prev->next = next;
            }
            prev = curr;
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 5: Using Vector for Storage

Time Complexity: O(n)
Space Complexity: O(k)

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (!head || k == 1) return head;

        vector<ListNode*> group(k);
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;
        ListNode* curr = head;

        while (curr) {
            // Store k nodes in vector
            int count = 0;
            ListNode* temp = curr;
            while (temp && count < k) {
                group[count] = temp;
                temp = temp->next;
                count++;
            }
            // If we have k nodes, reverse them
            if (count == k) {
                // Connect in reverse order
                for (int i = k - 1; i >= 0; i--) {
                    tail->next = group[i];
                    tail = tail->next;
                }
                tail->next = temp; // Connect to next segment
                curr = temp;
            } else {
                // Connect remaining nodes as they are
                tail->next = curr;
                break;
            }
        }
        return dummy->next;
    }
};
```

Solution 6: Modular Iterative Approach

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

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (!head || k == 1) return head;

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* groupPrev = dummy;

        while (true) {
            // Check if we have k nodes remaining
            ListNode* kth = getKthNode(groupPrev, k);
            if (!kth) break;

            ListNode* groupNext = kth->next;

            // Reverse current group
            ListNode* prev = kth->next;
            ListNode* curr = groupPrev->next;

            while (curr != groupNext) {
                ListNode* next = curr->next;
                curr->next = prev;
                prev = curr;
                curr = next;
            }
      // Connect previous group to current reversed group
            ListNode* temp = groupPrev->next;
            groupPrev->next = kth;
            groupPrev = temp;
        }
        return dummy->next;
    }
private:
    ListNode* getKthNode(ListNode* curr, int k) {
        while (curr && k > 0) {
            curr = curr->next;
            k--;
        }
        return curr;
```

Solution 7: Enhanced Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n/k)

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* curr = head;
        int count = 0;

        // First, check if we have k nodes
        while (curr && count < k) {
            curr = curr->next;
            count++;
        }

        if (count == k) {
          // Reverse first k nodes and get the new head
          ListNode* reversedHead = reverseFirstK(head, k);

            // head is now the tail of reversed group
            // Recursively process the remaining list
            head->next = reverseKGroup(curr, k);

            return reversedHead;
        }
        // Return head as it is if less than k nodes
        return head;
    }
private:
    ListNode* reverseFirstK(ListNode* head, int k) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        for (int i = 0; i < k; i++) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

61 : Problem : **Rotate List**

Given the head of a linked list, rotate the list to the right by k places.

Example 1:
Input: `head = [1,2,3,4,5], k = 2`
Output: `[4,5,1,2,3]`
Explanation:
rotate 1 steps to the right: `[5,1,2,3,4]`
rotate 2 steps to the right: `[4,5,1,2,3]`

Example 2:
Input: `head = [0,1,2], k = 4`
Output: `[2,0,1]`
Explanation:
rotate 1 steps to the right: `[2,0,1]`
rotate 2 steps to the right: `[1,2,0]`
rotate 3 steps to the right: `[0,1,2]`
rotate 4 steps to the right: `[2,0,1]`

Solution 1: Two-Pointers (Optimal Approach)

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

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head;

        // Step 1: Calculate length of list
        ListNode* curr = head;
        int length = 1;
        while (curr->next) {
            curr = curr->next;
            length++;
        }

        // Step 2: Connect tail to head to make circular list
        curr->next = head;

        // Step 3: Calculate effective rotations needed
        k = k % length;
        int stepsToNewHead = length - k;

        // Step 4: Find new tail and new head
        ListNode* newTail = head;
        for (int i = 1; i < stepsToNewHead; i++) {
            newTail = newTail->next;
        }

        ListNode* newHead = newTail->next;
        newTail->next = nullptr;

        return newHead;
    }
};
```

Solution 2: Three-Pass Approach

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

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || k == 0) return head;

        // First pass: calculate length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Calculate effective rotations
        k = k % length;
        if (k == 0) return head;

        // Second pass: find the (length - k)th node
        ListNode* slow = head;
        ListNode* fast = head;

        for (int i = 0; i < k; i++) {
            fast = fast->next;
        }

        while (fast->next) {
            slow = slow->next;
            fast = fast->next;
        }

        // Third pass: rearrange pointers
        ListNode* newHead = slow->next;
        slow->next = nullptr;
        fast->next = head;
        return newHead;
    }
};
```

Solution 3: Stack-Based Approach

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

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || k == 0) return head;

        stack<ListNode*> st;
        ListNode* curr = head;
        int length = 0;

        // Push all nodes to stack and count length
        while (curr) {
            st.push(curr);
            curr = curr->next;
            length++;
        }

        k = k % length;
        if (k == 0) return head;

        // Pop k nodes to find new head and tail
        ListNode* newTail = nullptr;
        for (int i = 0; i < k; i++) {
            newTail = st.top();
            st.pop();
        }

        ListNode* newHead = newTail;
        ListNode* oldTail = st.top(); // Last node before rotation point

        // Find the original tail
        while (!st.empty()) {
            curr = st.top();
            st.pop();
        }
        curr->next = head; // Connect original tail to head
        oldTail->next = nullptr; // Break the cycle

        return newHead;
    }
};
```

Solution 4: Array Storage Approach

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

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head) return head;

        vector<ListNode*> nodes;
        ListNode* curr = head;

        // Store all nodes in array
        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }
        int length = nodes.size();
        k = k % length;
        if (k == 0) return head;

        // Calculate new head index
        int newHeadIndex = length - k;

        // Rearrange pointers
        nodes[length - 1]->next = nodes[0]; // Connect last to first
        nodes[newHeadIndex - 1]->next = nullptr; // Break the cycle

        return nodes[newHeadIndex];
```

Solution 5: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || k == 0) return head;

        // First, calculate length
        int length = getLength(head);
        k = k % length;

        if (k == 0) return head;

        return rotateHelper(head, k);
    }

private:
    int getLength(ListNode* head) {
        int length = 0;
        while (head) {
            length++;
            head = head->next;
        }
        return length;
    }

    ListNode* rotateHelper(ListNode* head, int k) {
        if (k == 0) return head;

        // Find the last node and second last node
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr->next) {
            prev = curr;
            curr = curr->next;
        }

        // Move last node to front
        prev->next = nullptr;
        curr->next = head;

        // Recursively rotate remaining k-1 times
        return rotateHelper(curr, k - 1);
    }
};
```

Solution 6: Modular Arithmetic with Single Pass

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

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head;

        ListNode* curr = head;
        int length = 1;

        // Calculate length and find tail
        while (curr->next) {
            curr = curr->next;
            length++;
        }

        // Make circular
        curr->next = head;

        // Calculate break point
        k = length - (k % length);

        // Find new tail
        for (int i = 0; i < k; i++) {
            curr = curr->next;
        }

        // Break the circle
        head = curr->next;
        curr->next = nullptr;

        return head;
    }
};
```

Solution 7: Two-Pointer with Gap

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

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || k == 0) return head;

        // Calculate length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        k = k % length;
        if (k == 0) return head;

        // Use two pointers with k gap
        ListNode* slow = head;
        ListNode* fast = head;
```

```
        // Move fast k steps ahead
        for (int i = 0; i < k; i++) {
            fast = fast->next;
        }

        // Move both until fast reaches end
        while (fast->next) {
            slow = slow->next;
            fast = fast->next;
        }

        // Rearrange pointers
        ListNode* newHead = slow->next;
        slow->next = nullptr;
        fast->next = head;

        return newHead;
    }
};
```

Solution 8: Brute Force (Rotate One by One)

Time Complexity: O(k × n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || k == 0) return head;

        // Calculate length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        k = k % length;

        // Rotate k times (one rotation at a time)
        for (int i = 0; i < k; i++) {
            head = rotateOnce(head);
        }

        return head;
    }

private:
    ListNode* rotateOnce(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* prev = nullptr;
        ListNode* curr = head;

        // Find last and second last nodes
        while (curr->next) {
            prev = curr;
            curr = curr->next;
        }

        // Move last node to front
        prev->next = nullptr;
        curr->next = head;

        return curr;
    }
};
```

82 : Problem :Remove Duplicates from Sorted List II

Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the modified linked list.

Example 1:
Input: head = [1,2,3,3,4,4,5]
Output: [1,2,5]

Example 2:
Input: head = [1,1,1,2,3]
Output: [2,3]

Solution 1. Iterative Approach with Dummy Node

TimeComplexity: O(n)
Space Complexity: O(1)
Idea: Use a dummy node to handle edge cases and two pointers to track duplicates.

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;
        ListNode* curr = head;

        while(curr) {
            bool duplicate = false;
            while(curr->next && curr->val == curr->next->val) {
                duplicate = true;
                curr = curr->next;
            }

            if(duplicate) {
                prev->next = curr->next;
            } else {
                prev = prev->next;
            }
            curr = curr->next;
        }
        return dummy->next;
    }
```

Solution 2. Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) (recursion stack)
Idea: Recursively process the list, skipping duplicate nodes.

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if(!head || !head->next) return head;

        if(head->val == head->next->val) {
            int val = head->val;
            while(head && head->val == val) {
                head = head->next;
            }
            return deleteDuplicates(head);
        }

        head->next = deleteDuplicates(head->next);
        return head;
    }
};
```

Solution 3. Hash Map Approach

Time Complexity: O(n)
Space Complexity: O(n)
Idea: Use a hash map to count occurrences and then rebuild the list.

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        unordered_map<int,int> count;
        ListNode* curr = head;

        while(curr) {
            count[curr->val]++;
            curr = curr->next;
        }

        ListNode* dummy = new ListNode(0);
        ListNode* newCurr = dummy;
        curr = head;

        while(curr) {
            if(count[curr->val] == 1) {
                newCurr->next = new ListNode(curr->val);
                newCurr = newCurr->next;
            }
            curr = curr->next;
        }
        return dummy->next;
    }
};
```

83 : Problem :Remove Duplicates from Sorted List

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

Example 1:
Input: `head = [1,1,2]`
Output: `[1,2]`

Example 2:
Input: `head = [1,1,2,3,3]`
Output: `[1,2,3]`

Example 3:
Input: `head = [1,1,1]`
Output: `[1]`

Solution 1: Iterative Approach (Recommended)

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* curr = head;

        while (curr && curr->next) {
            if (curr->val == curr->next->val) {
                // Remove duplicate
                ListNode* duplicate = curr->next;
                curr->next = curr->next->next;
                delete duplicate;
            } else {
                // Move to next distinct element
                curr = curr->next;
            }
        }

        return head;
    }
};
```

Solution 2: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head || !head->next) return head;

        // Recursively process the rest of the list
        head->next = deleteDuplicates(head->next);

        // If current node has same value as next, skip current node
        if (head->val == head->next->val) {
            ListNode* next = head->next;
            delete head;
            return next;
        }
        return head;
    }
};
```

Solution 3: Two-Pointer Approach

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* slow = head;
        ListNode* fast = head->next;

        while (fast) {
            if (slow->val != fast->val) {
                // Found distinct element
                slow->next = fast;
                slow = slow->next;
            }
            fast = fast->next;
        }

        // Terminate the list
        slow->next = nullptr;

        return head;
    }
};
```

Solution 4: Using Dummy Node

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return head;

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;
        ListNode* curr = head;

        while (curr) {
            // Skip all duplicates
            while (curr->next && curr->val == curr->next->val)
{
                curr = curr->next;
            }

            // Connect prev to current (distinct element)
            prev->next = curr;
            prev = prev->next;
            curr = curr->next;
        }

        ListNode* result = dummy->next;
        delete dummy;
        return result;
    }
};
```

Solution 5: In-Place Modification with Pointer

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* current = head;

        while (current && current->next) {
            if (current->val == current->next->val) {
                // Skip the duplicate node
                current->next = current->next->next;
            } else {
                // Move to next node only if distinct
                current = current->next;
            }
        }
        return head;
    }
};
```

Solution 6: Functional Recursive Approach

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        return removeDuplicates(head);
    }

private:
    ListNode* removeDuplicates(ListNode* node) {
        if (!node || !node->next) return node;

        ListNode* nextDistinct =
removeDuplicates(node->next);

        if (node->val == nextDistinct->val) {
            delete node;
            return nextDistinct;
        }
        node->next = nextDistinct;
        return node;
    }
};
```

Solution 7: Iterative with Previous Pointer

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* prev = head;
        ListNode* curr = head->next;

        while (curr) {
            if (prev->val == curr->val) {
                // Remove duplicate
                prev->next = curr->next;
                delete curr;
                curr = prev->next;
            } else {
                // Move both pointers
                prev = curr;
                curr = curr->next;
            }
        }
        return head;
    }
};
```

Solution 8: Using Set (Alternative Approach)

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

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return head;

        unordered_set<int> seen;
        ListNode* curr = head;
        ListNode* prev = nullptr;

        while (curr) {
            if (seen.find(curr->val) != seen.end()) {
                // Duplicate found, remove it
                prev->next = curr->next;
                delete curr;
                curr = prev->next;
            } else {
                // New value, add to set and move forward
                seen.insert(curr->val);
                prev = curr;
                curr = curr->next;
            }
        }
        return head;
    }
};
```

86 : Problem : **Partition List**

Given the head of a linked list and a value x, partition the list such that all nodes less than x come before nodes greater than or equal to x. You should preserve the original relative order of the nodes in each of the two partitions.

Example 1:
Input: head = [1,4,3,2,5,2], x = 3
Output: [1,2,2,4,3,5]

Example 2:
Input: head = [2,1], x = 2
Output: [1,2]

Solution 1. Two Separate Lists Approach

Time Complexity: O(n)
Space Complexity: O(1)
Idea: Create two separate lists for nodes less than x and nodes >= x, then merge them.

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode beforeHead(0);
        ListNode afterHead(0);
        ListNode* before = &beforeHead;
        ListNode* after = &afterHead;

        while(head) {
            if(head->val < x) {
                before->next = head;
                before = before->next;
            } else {
```

```
            after->next = head;
            after = after->next;
        }
        head = head->next;
    }
    after->next = nullptr;
    before->next = afterHead.next;
    return beforeHead.next;
  }
};
```

Solution 2. In-place Reordering Approach

Time Complexity: O(n)
Space Complexity: O(1)
Idea: Modify the list in-place by moving nodes to their correct positions.

```
class Solution {
public:
  ListNode* partition(ListNode* head, int x) {
    ListNode* dummy = new ListNode(0);
    dummy->next = head;
    ListNode* prev = dummy;
    ListNode* curr = head;

    while(curr && curr->val < x) {
        prev = curr;
        curr = curr->next;
    }

    ListNode* insertPos = prev;

    while(curr) {
        if(curr->val < x) {
            prev->next = curr->next;
            curr->next = insertPos->next;
            insertPos->next = curr;
            insertPos = insertPos->next;
            curr = prev->next;
        } else {
            prev = curr;
            curr = curr->next;
        }
    }

    return dummy->next;
 }
```

Solution 3. Vector Sorting Approach

Time Complexity: O(n log n)
Space Complexity: O(n)
Idea: Convert list to vector, sort by partition, then rebuild list.

```
class Solution {
public:
  ListNode* partition(ListNode* head, int x) {
    vector<ListNode*> nodes;
    while(head) {
        nodes.push_back(head);
        head = head->next;
    }

    stable_sort(nodes.begin(), nodes.end(),
[x](ListNode* a, ListNode* b) {
        return (a->val < x) && (b->val >= x);
    });

    ListNode dummy(0);
    ListNode* curr = &dummy;
    for(auto node : nodes) {
        curr->next = node;
        curr = curr->next;
    }
    curr->next = nullptr;

    return dummy.next;
  }
};
```

92 : Problem : **Reverse Linked List II**

Given the head of a singly linked list and two integers `left` and `right` (where `left <= right`), reverse the nodes from position `left` to position `right` and return the modified list.

Example:

Input: head = [1,2,3,4,5], left = 2, right = 4
Output: [1,4,3,2,5]

Solution 1: Iterative Approach (Standard)

1. Traverse to the node before the `left` position
2. Reverse the sublist from `left` to `right`
3. Reconnect the reversed sublist back to the main list

```
ListNode* reverseBetween(ListNode* head, int left, int
right) {
    if (!head || left == right) return head;

    ListNode dummy(0);
    dummy.next = head;
    ListNode* prev = &dummy;

    // Move to the node before left
    for (int i = 1; i < left; i++) {
        prev = prev->next;
    }
    // Reverse the sublist
    ListNode* curr = prev->next;
    ListNode* next = nullptr;
    ListNode* tail = curr; // Will be the tail of reversed sublist
    for (int i = left; i <= right; i++) {
        next = curr->next;
        curr->next = prev->next;
        prev->next = curr;
        curr = next;
    }
    // Connect the tail to remaining nodes
    tail->next = curr;

    return dummy.next;
}
```

Solution 2: Recursive Approach

1. Base case: if left == 1, reverse first `right` nodes
2. Otherwis recursively move to next node until left == 1
3. Reverse the required portion and reconnect

```
ListNode* reverseBetween(ListNode* head, int left, int
right) {
    if (left == 1) {
        return reverseN(head, right);
    }
    head->next = reverseBetween(head->next, left-1,
right-1);
    return head;
}

ListNode* reverseN(ListNode* head, int n) {
    if (n == 1) return head;
    ListNode* new_head = reverseN(head->next, n-1);
    ListNode* next = head->next->next;
    head->next->next = head;
    head->next = next;
    return new_head;
}
```

Solution 3: Using Stack

1. Push nodes from `left` to `right` onto a stack
2. Pop nodes from stack to rebuild the reversed portion

```
ListNode* reverseBetween(ListNode* head, int left, int
right) {
    if (!head || left == right) return head;

    stack<ListNode*> st;
    ListNode dummy(0);
    dummy.next = head;
    ListNode* curr = &dummy;

    // Move to node before left
    for (int i = 1; i < left; i++) {
        curr = curr->next;
    }

    // Push nodes to stack
    ListNode* start = curr;
    curr = curr->next;
    for (int i = left; i <= right; i++) {
        st.push(curr);
        curr = curr->next;
    }
    ListNode* end = curr;

    // Pop from stack to reverse
    while (!st.empty()) {
        start->next = st.top();
        st.pop();
        start = start->next;
    }
    start->next = end;
    return dummy.next;
}
```

Solution 4: Partial Reversal with Pointers

1. Identify the sublist to reverse
2. Reverse it while keeping track of surrounding nodes
3. Reconnect the reversed portion

```
ListNode* reverseBetween(ListNode* head, int left, int
right) {
    if (!head || left == right) return head;

    ListNode* prev = nullptr;
    ListNode* curr = head;

    // Move to left position
    for (int i = 1; i < left; i++) {
        prev = curr;
        curr = curr->next;
    }

    ListNode* con = prev; // Node before reversed portion
    ListNode* tail = curr; // Will be tail of reversed portion

    // Reverse the sublist
    ListNode* next = nullptr;
    for (int i = left; i <= right; i++) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    // Reconnect
    if (con) {
        con->next = prev;
    } else {
        head = prev;
    }
    tail->next = curr;
    return head;
}
```

138 : Problem: Copy List with Random Pointer

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a deep copy of the list. The deep copy should consist of exactly `n` new nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

Example 1:
Input: `head = [[7,null],[13,0],[11,4],[10,2],[1,0]]`
Output: `[[7,null],[13,0],[11,4],[10,2],[1,0]]`

Example 2:
Input: `head = [[1,1],[2,1]]`
Output: `[[1,1],[2,1]]`

Example 3:
Input: `head = [[3,null],[3,0],[3,null]]`
Output: `[[3,null],[3,0],[3,null]]`

Solution 1: Hash Map with Two Passes (Most Intuitive)

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        unordered_map<Node*, Node*> mapping;

        // First pass: create all new nodes and store mapping
        Node* curr = head;
        while (curr) {
            mapping[curr] = new Node(curr->val);
            curr = curr->next;
        }

        // Second pass: assign next and random pointers
        curr = head;
        while (curr) {
            mapping[curr]->next = mapping[curr->next];
            mapping[curr]->random = mapping[curr->random];
            curr = curr->next;
        }
        return mapping[head];
    }
};
```

Solution 2: Interweaving Nodes (O(1) Space)

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        // Step 1: Create interweaved list: A->A'->B->B'->C->C'
        Node* curr = head;
        while (curr) {
            Node* copy = new Node(curr->val);
            copy->next = curr->next;
            curr->next = copy;
            curr = copy->next;
        }

        // Step 2: Assign random pointers for copies
        curr = head;
        while (curr) {
            if (curr->random) {
                curr->next->random = curr->random->next;
            }
            curr = curr->next->next;
        }

        // Step 3: Separate the interweaved list
        Node* original = head;
        Node* copyHead = head->next;
        Node* copy = copyHead;

        while (original) {
            original->next = original->next->next;
            if (copy->next) {
                copy->next = copy->next->next;
            }
            original = original->next;
            copy = copy->next;
        }

        return copyHead;
    }
};
```

Solution 3: Recursive with Hash Map

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        unordered_map<Node*, Node*> visited;
        return copyRandomListHelper(head, visited);
    }

private:
    Node* copyRandomListHelper(Node* node,
unordered_map<Node*, Node*>& visited) {
        if (!node) return nullptr;

        // If node already copied, return the copy
        if (visited.find(node) != visited.end()) {
            return visited[node];
        }

        // Create new node
        Node* newNode = new Node(node->val);
        visited[node] = newNode;
        // Recursively copy next and random
        newNode->next =
copyRandomListHelper(node->next, visited);
        newNode->random =
copyRandomListHelper(node->random, visited);

        return newNode;
    }
};
```

Solution 4: Iterative with Visited Dictionary

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        unordered_map<Node*, Node*> visited;
        Node* oldNode = head;
        Node* newNode = new Node(head->val);
        visited[oldNode] = newNode;

        while (oldNode) {
            // Copy next pointer
            if (oldNode->next) {
                if (visited.find(oldNode->next) == visited.end()) {
                    visited[oldNode->next] = new
Node(oldNode->next->val);
                }
                newNode->next = visited[oldNode->next];
            }

            // Copy random pointer
            if (oldNode->random) {
                if (visited.find(oldNode->random) ==
visited.end()) {
                    visited[oldNode->random] = new
Node(oldNode->random->val);
                }
                newNode->random =
visited[oldNode->random];
            }

            oldNode = oldNode->next;
            newNode = newNode->next;
        }

        return visited[head];
    }
};
```

Solution 5: Using Vector for Storage

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        vector<Node*> originalNodes;
        vector<Node*> copyNodes;

        // First pass: create copy nodes and store both lists
        Node* curr = head;
        while (curr) {
            originalNodes.push_back(curr);
            copyNodes.push_back(new Node(curr->val));
            curr = curr->next;
        }

        // Second pass: assign next and random pointers
        for (int i = 0; i < originalNodes.size(); i++) {
            // Assign next pointer
            if (i < originalNodes.size() - 1) {
                copyNodes[i]->next = copyNodes[i + 1];
            }

            // Assign random pointer
            if (originalNodes[i]->random) {
                // Find index of random node in original list
                auto it = find(originalNodes.begin(),
originalNodes.end(), originalNodes[i]->random);
                int randomIndex =
distance(originalNodes.begin(), it);
                copyNodes[i]->random =
copyNodes[randomIndex];
            }
        }
        return copyNodes[0];
    }
};
```

Solution 6: Two Pass with Array and Map

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        vector<Node*> copyNodes;
        unordered_map<Node*, int> nodeToIndex;

        // First pass: create copy nodes and map original nodes to indices
        Node* curr = head;
        int index = 0;
        while (curr) {
            copyNodes.push_back(new Node(curr->val));
            nodeToIndex[curr] = index;
            curr = curr->next;
            index++;
        }

        // Second pass: assign next and random pointers
        curr = head;
        for (int i = 0; i < copyNodes.size(); i++) {
            // Assign next pointer
            if (i < copyNodes.size() - 1) {
                copyNodes[i]->next = copyNodes[i + 1];
            }
            // Assign random pointer
            if (curr->random) {
                int randomIndex = nodeToIndex[curr->random];
                copyNodes[i]->random =
copyNodes[randomIndex];
            }
            curr = curr->next;
        }
        return copyNodes[0];
    }
};
```

Solution 7: Modified Interweaving (Alternative)

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        // Step 1: Create copy nodes and insert after originals
        Node* curr = head;
        while (curr) {
            Node* copy = new Node(curr->val);
            copy->next = curr->next;
            curr->next = copy;
            curr = copy->next;
        }
        // Step 2: Assign random pointers
        curr = head;
        while (curr) {
            if (curr->random) {
                curr->next->random = curr->random->next;
            }
            curr = curr->next->next;
        }
        // Step 3: Extract copy list and restore original
        Node* copyHead = head->next;
        Node* original = head;
        Node* copy = copyHead;

        while (original) {
            original->next = original->next->next;
            if (copy->next) {
                copy->next = copy->next->next;
            }
            original = original->next;
            copy = copy->next;
        }
        return copyHead;
    }
};
```

Solution 8: BFS-like Approach

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return nullptr;

        unordered_map<Node*, Node*> visited;
        queue<Node*> q;

        Node* copyHead = new Node(head->val);
        visited[head] = copyHead;
        q.push(head);

        while (!q.empty()) {
            Node* curr = q.front();
            q.pop();

            // Process next pointer
            if (curr->next) {
                if (visited.find(curr->next) == visited.end()) {
                    visited[curr->next] = new
Node(curr->next->val);
                    q.push(curr->next);
                }
                visited[curr]->next = visited[curr->next];
            }

            // Process random pointer
            if (curr->random) {
                if (visited.find(curr->random) == visited.end()) {
                    visited[curr->random] = new
Node(curr->random->val);
                    q.push(curr->random);
                }
                visited[curr]->random = visited[curr->random];
            }
        }

        return copyHead;
    }
};
```

141 : Problem : **Linked List Cycle**

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer.

Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

Example 1:
Input: `head = [3,2,0,-4], pos = 1`
Output: `true`
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:
Input: `head = [1,2], pos = 0`
Output: `true`
Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:
Input: `head = [1], pos = -1`
Output: `false`
Explanation: There is no cycle in the linked list.

Solution 1: Floyd's Cycle-Finding Algorithm **(Two Pointers)**

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

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (!head || !head->next) return false;

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;

            if (slow == fast) {
                return true;
            }
        }

        return false;
    }
};
```

Solution 2: Hash Set Approach

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

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        unordered_set<ListNode*> visited;
        ListNode* curr = head;

        while (curr) {
            if (visited.find(curr) != visited.end()) {
                return true;
            }
            visited.insert(curr);
            curr = curr->next;
        }
        return false;
    }
};
```

Solution 3: Marking Nodes (Modification Approach)

Time Complexity: O(n)
Space Complexity: O(1) - but modifies the list

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* curr = head;

        while (curr) {
            if (curr->val == INT_MAX) {
                return true;
            }
            curr->val = INT_MAX; // Mark as visited
            curr = curr->next;
        }
        return false;
    }
};
```

Solution 4: Reverse List Approach

Time Complexity: O(n)
Space Complexity: O(1) - but modifies the list

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (!head || !head->next) return false;

        ListNode* reversed = reverseList(head);

        // If there was a cycle, after reversal we'll get back to original head
        if (head == reversed) {
            return true;
        }
        return false;
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

       while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

Solution 5: Recursive with Hash Set

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack + hash set

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        unordered_set<ListNode*> visited;
        return hasCycleHelper(head, visited);
    }
private:
    bool hasCycleHelper(ListNode* node,
unordered_set<ListNode*>& visited) {
        if (!node) return false;

        if (visited.find(node) != visited.end()) {
            return true;
        }
        visited.insert(node);
        return hasCycleHelper(node->next, visited);
    }
};
```

Solution 6: Two Pointers with Different Speeds

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

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (!head || !head->next) return false;

        ListNode* slow = head;
        ListNode* fast = head->next;

        while (slow != fast) {
            if (!fast || !fast->next) {
                return false;
            }
            slow = slow->next;
            fast = fast->next->next;
        }
        return true;
    }
};
```

Solution 7: Using Unique Address Trick

Time Complexity: O(n)
Space Complexity: O(1) - but modifies the list

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* curr = head;

        while (curr) {
     // Check if we've visited this node by checking if next points to itself
            if (curr->next == curr) {
                return true;
            }
            ListNode* next = curr->next;
            curr->next = curr; // Mark current node as visited
            curr = next;
        }
       return false;
    }
};
```

Solution 8: Brent's Algorithm **(Improved Cycle Detection)**

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

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (!head || !head->next) return false;

        ListNode* slow = head;
        ListNode* fast = head;
        int steps = 0;
        int limit = 2;

        while (fast && fast->next) {
            fast = fast->next;
            steps++;

            if (slow == fast) {
                return true;
            }

            if (steps == limit) {
                steps = 0;
                limit *= 2;
                slow = fast;
            }
        }
        return false;
    }
};
```

Solution 9: Length Counting Approach

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

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if (!head) return false;

        int count = 0;
        ListNode* curr = head;

        while (curr) {
            count++;
            if (count > 10000) { // Assuming max nodes based
on constraints
                return true;
            }
            curr = curr->next;
        }

        return false;
    }
};
```

Solution 10: Dummy Node Approach

Time Complexity: O(n)
Space Complexity: O(1) - but modifies the list

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = head;

        while (curr) {
            if (curr->next == dummy) {
                delete dummy;
                return true;
            }

            ListNode* next = curr->next;
            curr->next = dummy; // Point to dummy to mark as visited
            curr = next;
        }

        delete dummy;
        return false;
    }
};
```

142 : Problem: **Linked List Cycle II**

Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer.

Example 1:
Input: `head = [3,2,0,-4], pos = 1`
Output: `tail connects to node index 1`
Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:
Input: `head = [1,2], pos = 0`
Output: `tail connects to node index 0`
Explanation: There is a cycle in the linked list, where tail connects to the first node.

Example 3:
Input: `head = [1], pos = -1`
Output: `no cycle`
Explanation: There is no cycle in the linked list.

Solution 1: Floyd's Cycle Detection (Two Pointers)

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

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (!head || !head->next) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;

        // Step 1: Detect if cycle exists
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;

            if (slow == fast) {
                // Step 2: Find the cycle entry point
                ListNode* entry = head;
                while (entry != slow) {
                    entry = entry->next;
                    slow = slow->next;
                }
                return entry;
            }
        }
        return nullptr;
    }
};
```

Solution 2: Hash Set Approach

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

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        unordered_set<ListNode*> visited;
        ListNode* curr = head;

        while (curr) {
            if (visited.find(curr) != visited.end()) {
                return curr;
            }
            visited.insert(curr);
            curr = curr->next;
        }
        return nullptr;
    }
};
```

Solution 3: Marking Nodes with Special Value

Time Complexity: O(n)
Space Complexity: O(1) - but modifies node values

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* curr = head;

        while (curr) {
            if (curr->val == INT_MAX) {
                return curr;
            }
            curr->val = INT_MAX; // Mark as visited
            curr = curr->next;
        }
        return nullptr;
    }
};
```

Solution 4: Node Self-Marking Approach

Time Complexity: O(n)
Space Complexity: O(1) - but modifies the list structure

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* curr = head;

        while (curr) {
            // Check if current node points to itself (marked)
            if (curr->next == curr) {
                return curr;
            }
            ListNode* next = curr->next;
            curr->next = curr; // Mark current node by pointing
to itself
            curr = next;
        }
        return nullptr;
    }
};
```

Solution 5: Length-Based Approach

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

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (!head || !head->next) return nullptr;

        // Step 1: Detect cycle and find meeting point
        ListNode* slow = head;
        ListNode* fast = head;
        bool hasCycle = false;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;

            if (slow == fast) {
                hasCycle = true;
                break;
            }
        }

        if (!hasCycle) return nullptr;

        // Step 2: Calculate cycle length
        int cycleLength = 1;
        ListNode* temp = slow->next;
        while (temp != slow) {
            cycleLength++;
            temp = temp->next;
        }

        // Step 3: Find cycle entry using two pointers with cycle length gap
        ListNode* ptr1 = head;
        ListNode* ptr2 = head;

        // Move ptr2 cycleLength steps ahead
        for (int i = 0; i < cycleLength; i++) {
            ptr2 = ptr2->next;
        }

        // Move both until they meet at cycle entry
        while (ptr1 != ptr2) {
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
        }
        return ptr1;
    }
};
```

Solution 6: Recursive with Hash Set

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        unordered_set<ListNode*> visited;
        return detectCycleHelper(head, visited);
    }

private:
    ListNode* detectCycleHelper(ListNode* node,
unordered_set<ListNode*>& visited) {
        if (!node) return nullptr;
        if (visited.find(node) != visited.end()) {
            return node;
        }

        visited.insert(node);
        return detectCycleHelper(node->next, visited);
    }
};
```

Solution 7: Dummy Node Marker

Time Complexity: O(n)
Space Complexity: O(1) - but uses extra node

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = head;

        while (curr) {
            if (curr->next == dummy) {
                delete dummy;
                return curr;
            }

            ListNode* next = curr->next;
            curr->next = dummy; // Mark by pointing to dummy
            curr = next;
        }
        delete dummy;
        return nullptr;
    }
};
```

Solution 8: Brent's Algorithm

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

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (!head || !head->next) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;
        int power = 1;
        int length = 1;

        while (fast && fast->next) {
            fast = fast->next;

            if (slow == fast) {
                // Found cycle, now find start
                ListNode* ptr1 = head;
                ListNode* ptr2 = slow;
```

```
            while (ptr1 != ptr2) {
                ptr1 = ptr1->next;
                ptr2 = ptr2->next;
            }
            return ptr1;
        }
        if (length == power) {
            power *= 2;
            length = 0;
            slow = fast;
        }
        length++;
    }
    return nullptr;
  }
};
```

Solution 9: Two-Pass Floyd's Algorithm

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

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        // First pass: detect if cycle exists
        ListNode* meetingPoint = getMeetingPoint(head);
        if (!meetingPoint) return nullptr;

        // Second pass: find cycle entry
        ListNode* ptr1 = head;
        ListNode* ptr2 = meetingPoint;

        while (ptr1 != ptr2) {
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
        }
        return ptr1;
    }
private:
    ListNode* getMeetingPoint(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;

            if (slow == fast) {
                return slow;
            }
        }
        return nullptr;
    }
};
```

Solution 10: Visited Flag in Node Structure (If allowed)

Time Complexity: O(n)
Space Complexity: O(1) - but modifies node structure

```
struct ListNode {
    int val;
    ListNode *next;
    bool visited;
    ListNode(int x) : val(x), next(nullptr), visited(false) {}
};

class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* curr = head;

        while (curr) {
            if (curr->visited) {
                return curr;
            }
            curr->visited = true;
            curr = curr->next;
        }

        return nullptr;
    }
};
```

Mathematical Proof of Floyd's Algorithm:

Let:
- `L1` = distance from head to cycle entry
- `L2` = distance from cycle entry to meeting point
- `C` = length of cycle

When slow and fast meet:
- Slow has moved: `L1 + L2`
- Fast has moved: `L1 + L2 + n*C` (n complete cycles)

Since fast moves twice as fast:
`2(L1 + L2) = L1 + L2 + n*C`
`L1 + L2 = n*C`
`L1 = n*C - L2`

This means distance from head to entry equals distance from meeting point to entry (going around cycle).

143 : Problem : **Reorder List**

You are given the head of a singly linked-list. The list can be represented as:

L0 → L1 → … → Ln-1 → Ln

Reorder the list to be in the form:

L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → …

You may not modify the values in the list's nodes, only nodes themselves may be changed.

Example 1:
Input: `head = [1,2,3,4]`
Output: `[1,4,2,3]`

Example 2:
Input: `head = [1,2,3,4,5]`
Output: `[1,5,2,4,3]`

Solution 1: Three-Step Approach (Find Middle, Reverse, Merge)

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        // Step 1: Find the middle of the list
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Step 2: Reverse the second half
        ListNode* second = reverseList(slow->next);
        slow->next = nullptr; // Break the list into two parts
        ListNode* first = head;

        // Step 3: Merge the two lists alternately
        while (second) {
            ListNode* temp1 = first->next;
            ListNode* temp2 = second->next;

            first->next = second;
            second->next = temp1;

            first = temp1;
            second = temp2;
        }
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

Solution 2: Using Stack

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        stack<ListNode*> st;
        ListNode* curr = head;
        int length = 0;

        // Push all nodes to stack and count length
        while (curr) {
            st.push(curr);
            curr = curr->next;
            length++;
        }

        curr = head;
        int count = 0;

        // Reorder by alternating between beginning and end
        while (count < length / 2) {
            ListNode* next = curr->next;
            ListNode* end = st.top();
            st.pop();

            curr->next = end;
            end->next = next;

            curr = next;
            count++;
        }
        curr->next = nullptr;
    }
};
```

Solution 3: Using Vector/Array

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        vector<ListNode*> nodes;
        ListNode* curr = head;

        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        int left = 0, right = nodes.size() - 1;

        // Reorder using two pointers
        while (left < right) {
            nodes[left]->next = nodes[right];
            left++;

            if (left >= right) break;

            nodes[right]->next = nodes[left];
            right--;
        }
        nodes[left]->next = nullptr;
    }
};
```

Solution 4: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next || !head->next->next) return;

        // Find the second last node
        ListNode* secondLast = head;
        while (secondLast->next->next) {
            secondLast = secondLast->next;
        }

        // Move the last node after head
        ListNode* last = secondLast->next;
        secondLast->next = nullptr;

        ListNode* next = head->next;
        head->next = last;
        last->next = next;

        // Recursively reorder the remaining list
        reorderList(next);
    }
};
```

Solution 5: Two-Pointer with Deque

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        deque<ListNode*> dq;
        ListNode* curr = head;

        // Add all nodes to deque
        while (curr) {
            dq.push_back(curr);
            curr = curr->next;
        }

        ListNode* dummy = new ListNode(0);
        curr = dummy;
        bool fromFront = true;

       // Alternate between front and back
        while (!dq.empty()) {
            if (fromFront) {
                curr->next = dq.front();
                dq.pop_front();
            } else {
                curr->next = dq.back();
                dq.pop_back();
            }
            curr = curr->next;
            fromFront = !fromFront;
        }
        curr->next = nullptr;
        delete dummy;
    }
};
```

Solution 6: In-Place with Length Calculation

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        // Calculate length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Find the middle node
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Reverse second half
        ListNode* prev = nullptr;
        ListNode* second = slow->next;
        slow->next = nullptr;

        while (second) {
            ListNode* next = second->next;
            second->next = prev;
            prev = second;
            second = next;
        }

        // Merge two halves
        ListNode* first = head;
        second = prev;

        while (second) {
            ListNode* next1 = first->next;
            ListNode* next2 = second->next;

            first->next = second;
            second->next = next1;

            first = next1;
            second = next2;
        }
    }
};
```

Solution 7: Using Slow-Fast Pointer with Explicit Middle

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        // Step 1: Find the middle using slow-fast pointer
        ListNode* slow = head;
        ListNode* fast = head->next;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Step 2: Reverse the second half
        ListNode* second = slow->next;
        slow->next = nullptr;
        second = reverse(second);

        // Step 3: Merge the two lists
        ListNode* first = head;
        while (second) {
            ListNode* temp1 = first->next;
            ListNode* temp2 = second->next;

            first->next = second;
            second->next = temp1;

            first = temp1;
            second = temp2;
        }
    }

private:
    ListNode* reverse(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 8: Hybrid Approach with Vector and Two Pointers

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

```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (!head || !head->next) return;

        vector<ListNode*> arr;
        ListNode* curr = head;

        // Convert linked list to array
        while (curr) {
            arr.push_back(curr);
            curr = curr->next;
        }

        // Reorder using two pointers
        int i = 0, j = arr.size() - 1;
        while (i < j) {
            arr[i]->next = arr[j];
            i++;

            if (i >= j) break;

            arr[j]->next = arr[i];
            j--;
        }

        arr[i]->next = nullptr;
    }
};
```

147. Problem **Insertion Sort List**

Given the head of a singly linked list, sort the list using insertion sort and return the sorted list's head.

Example 1:
Input: head = [4,2,1,3]
Output: [1,2,3,4]

Example 2:
Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]

Solution 1. Standard Insertion Sort Implementation

Time Complexity: O(n^2)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode dummy(INT_MIN);
        ListNode* curr = head;
        while(curr) {
            ListNode* prev = &dummy;
            while(prev->next && prev->next->val < curr->val) {
                prev = prev->next;
            }
            ListNode* next = curr->next;
            curr->next = prev->next;
            prev->next = curr;
            curr = next;
        }

        return dummy.next;
    }
};
```

Solution 2. Optimized Insertion Sort with Tail Pointer

Time Complexity: O(n^2) but faster for nearly sorted lists
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode dummy(INT_MIN);
        ListNode* tail = &dummy;

        while(head) {
            if(tail->val <= head->val) {
                tail->next = head;
                tail = head;
                head = head->next;
            } else {
                ListNode* prev = &dummy;
                while(prev->next && prev->next->val <
head->val) {
                    prev = prev->next;
                }
                ListNode* next = head->next;
                head->next = prev->next;
                prev->next = head;
                head = next;
            }
        }
        tail->next = nullptr;
        return dummy.next;
    }
};
```

Solution 3. Convert to Vector and Sort

Time Complexity: O(n log n)
Space Complexity: O(n)

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        vector<int> nums;
        ListNode* curr = head;
        while(curr) {
            nums.push_back(curr->val);
            curr = curr->next;
        }
        sort(nums.begin(), nums.end());

        curr = head;
        for(int num : nums) {
            curr->val = num;
            curr = curr->next;
        }
        return head;
    }
};
```

148 : Problem : Sort List

Given the head of a linked list, return the list after sorting it in ascending order.

Example 1:
Input: `head = [4,2,1,3]`
Output: `[1,2,3,4]`

Example 2:
Input: `head = [-1,5,3,4,0]`
Output: `[-1,0,3,4,5]`

Example 3:
Input: `head = []`
Output: `[]`

Follow up: Can you sort the linked list in O(n log n) time and O(1) memory (i.e., constant space)?

Solution 1: Merge Sort (Top-Down Recursive)

Time Complexity: O(n log n)
Space Complexity: O(log n) - recursion stack

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        // Find the middle of the list
        ListNode* mid = getMid(head);
        ListNode* left = head;
        ListNode* right = mid->next;
        mid->next = nullptr; // Split the list

        // Recursively sort both halves
        left = sortList(left);
        right = sortList(right);

        // Merge the sorted halves
        return merge(left, right);
    }
private:
    ListNode* getMid(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head->next;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val <= l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }
        if (l1) curr->next = l1;
        if (l2) curr->next = l2;

        return dummy->next;
    }
};
```

Solution 2: Merge Sort (Bottom-Up Iterative)

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

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        int length = getLength(head);
        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        for (int step = 1; step < length; step *= 2) {
            ListNode* prev = dummy;
            ListNode* curr = dummy->next;

            while (curr) {
                ListNode* left = curr;
```

```
            ListNode* right = split(left, step);
            curr = split(right, step);
            prev = merge(left, right, prev);
        }
    }
    return dummy->next;
  }
private:
  int getLength(ListNode* head) {
    int length = 0;
    while (head) {
      length++;
      head = head->next;
    }
    return length;
  }

  ListNode* split(ListNode* head, int step) {
    if (!head) return nullptr;

    for (int i = 1; i < step && head->next; i++) {
      head = head->next;
    }

    ListNode* right = head->next;
    head->next = nullptr;
    return right;
  }

  ListNode* merge(ListNode* l1, ListNode* l2, ListNode*
prev) {
    ListNode* curr = prev;

    while (l1 && l2) {
      if (l1->val <= l2->val) {
        curr->next = l1;
        l1 = l1->next;
      } else {
        curr->next = l2;
        l2 = l2->next;
      }
      curr = curr->next;
    }

    if (l1) curr->next = l1;
    if (l2) curr->next = l2;

    while (curr->next) curr = curr->next;
    return curr;
  }
};
```

Solution 3: Quick Sort

Time Complexity: O(n log n) average, O(n²) worst case
Space Complexity: O(log n) - recursion stack

```
class Solution {
public:
  ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head;

    // Choose pivot (using first element)
    ListNode* pivot = head;
    ListNode* less = new ListNode(0);
    ListNode* equal = new ListNode(0);
    ListNode* greater = new ListNode(0);

    ListNode* lessTail = less;
    ListNode* equalTail = equal;
    ListNode* greaterTail = greater;

    ListNode* curr = head;

    // Partition the list
    while (curr) {
      if (curr->val < pivot->val) {
        lessTail->next = curr;
        lessTail = lessTail->next;
      } else if (curr->val == pivot->val) {
        equalTail->next = curr;
        equalTail = equalTail->next;
      } else {
        greaterTail->next = curr;
        greaterTail = greaterTail->next;
      }
      curr = curr->next;
    }

    // Terminate the lists
    lessTail->next = nullptr;
    equalTail->next = nullptr;
    greaterTail->next = nullptr;

    // Recursively sort less and greater parts
    less->next = sortList(less->next);
    greater->next = sortList(greater->next);

    // Concatenate: less + equal + greater
    ListNode* result = concatenate(less->next,
equal->next, greater->next);

    delete less;
    delete equal;
    delete greater;

    return result;
  }

private:
  ListNode* concatenate(ListNode* less, ListNode*
equal, ListNode* greater) {
    ListNode* dummy = new ListNode(0);
    ListNode* curr = dummy;

    if (less) {
      curr->next = less;
      while (curr->next) curr = curr->next;
    }

    if (equal) {
      curr->next = equal;
      while (curr->next) curr = curr->next;
    }

    if (greater) {
      curr->next = greater;
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
  }
};
```

Solution 4: Convert to Array, Sort, and Rebuild

Time Complexity: O(n log n)
Space Complexity: O(n)

```
class Solution {
public:
  ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head;

    vector<int> values;
    ListNode* curr = head;

    // Store all values in vector
    while (curr) {
      values.push_back(curr->val);
      curr = curr->next;
    }
    // Sort the values
    sort(values.begin(), values.end());

    // Rebuild the linked list
    curr = head;
    for (int val : values) {
      curr->val = val;
      curr = curr->next;
    }
    return head;
  }
};
```

Solution 5: Insertion Sort

Time Complexity: O(n²)
Space Complexity: O(1)

```
class Solution {
public:
  ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head;

    ListNode* dummy = new ListNode(0);
    ListNode* curr = head;

    while (curr) {
      ListNode* prev = dummy;
      ListNode* next = curr->next;

      // Find the insertion position
      while (prev->next && prev->next->val < curr->val)
{
        prev = prev->next;
      }
      // Insert current node
      curr->next = prev->next;
      prev->next = curr;
      curr = next;
    }
    return dummy->next;
  }
};
```

Solution 6: Using Priority Queue (Min-Heap)

Time Complexity: O(n log n)
Space Complexity: O(n)

```
class Solution {
public:
  ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head;

    priority_queue<int, vector<int>, greater<int>> pq;
    ListNode* curr = head;
    // Push all values to min-heap
    while (curr) {
      pq.push(curr->val);
      curr = curr->next;
    }
    // Rebuild the list with sorted values
    curr = head;
    while (!pq.empty()) {
      curr->val = pq.top();
      pq.pop();
      curr = curr->next;
    }
    return head;
  }
};
```

Solution 7: Bubble Sort

Time Complexity: O(n²)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        bool swapped;
        do {
            swapped = false;
            ListNode* curr = head;
            ListNode* prev = nullptr;

            while (curr && curr->next) {
                if (curr->val > curr->next->val) {
                    // Swap nodes
                    ListNode* next = curr->next;
                    curr->next = next->next;
                    next->next = curr;

                    if (prev) {
                        prev->next = next;
                    } else {
                        head = next;
                    }
                    prev = next;
                    swapped = true;
                } else {
                    prev = curr;
                    curr = curr->next;
                }
            }
        } while (swapped);

        return head;
    }
};
```

Solution 8: Selection Sort

Time Complexity: O(n²)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* curr = head;

        while (curr) {
            ListNode* minNode = curr;
            ListNode* temp = curr->next;

            // Find the minimum node in remaining list
            while (temp) {
                if (temp->val < minNode->val) {
                    minNode = temp;
                }
                temp = temp->next;
            }
            // Swap values
            if (minNode != curr) {
                swap(curr->val, minNode->val);
            }
            curr = curr->next;
        }
        return head;
    }
};
```

Solution 9: Optimized Quick Sort with Random Pivot

Time Complexity: O(n log n) average
Space Complexity: O(log n)

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        // Find tail for random pivot selection
        ListNode* tail = head;
        int length = 1;
        while (tail->next) {
            tail = tail->next;
            length++;
        }

        return quickSort(head, tail);
    }

private:
    ListNode* quickSort(ListNode* head, ListNode* tail) {
        if (!head || head == tail) return head;

        ListNode* pivot = partition(head, tail);

        if (pivot != head) {
            ListNode* temp = head;
            while (temp->next != pivot) {
                temp = temp->next;
            }
            temp->next = nullptr;
            head = quickSort(head, temp);
            temp = getTail(head);
            temp->next = pivot;
        }
        pivot->next = quickSort(pivot->next, tail);
        return head;
    }

    ListNode* partition(ListNode* head, ListNode* tail) {
        ListNode* pivot = tail;
        ListNode* i = head;
        ListNode* j = head;

        while (j != tail) {
            if (j->val < pivot->val) {
                swap(i->val, j->val);
                i = i->next;
            }
            j = j->next;
        }

        swap(i->val, pivot->val);
        return i;
    }

    ListNode* getTail(ListNode* head) {
        while (head && head->next) {
            head = head->next;
        }
        return head;
    }
};
```

160 : Problem : Intersection of Two Linked Lists

Given the heads of two singly linked-lists `headA` and `headB`, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return `null`.

Note: The linked lists must retain their original structure after the function returns.

Example 1:
Input: `intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3`
Output: `Intersected at '8'`
Explanation: The two lists intersect at node with value 8.

Example 2:
Input: `intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1`
Output: `Intersected at '2'`
Explanation: The two lists intersect at node with value 2.

Example 3:
Input: `intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2`
Output: `No intersection`
Explanation: The two lists do not intersect.

Solution 1: Two Pointers (Length Adjustment)

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (!headA || !headB) return nullptr;

        // Calculate lengths of both lists
        int lenA = getLength(headA);
        int lenB = getLength(headB);

        // Move the longer list forward by the difference
        ListNode* ptrA = headA;
        ListNode* ptrB = headB;

        if (lenA > lenB) {
            for (int i = 0; i < lenA - lenB; i++) {
                ptrA = ptrA->next;
            }
        } else {
            for (int i = 0; i < lenB - lenA; i++) {
                ptrB = ptrB->next;
            }
        }

        // Move both pointers until they meet or reach end
        while (ptrA && ptrB) {
            if (ptrA == ptrB) {
                return ptrA;
            }
            ptrA = ptrA->next;
            ptrB = ptrB->next;
        }

        return nullptr;
    }

private:
    int getLength(ListNode* head) {
        int length = 0;
        while (head) {
            length++;
            head = head->next;
        }
        return length;
    }
};
```

Solution 2: Two Pointers (Cycle Detection Approach)

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        if (!headA || !headB) return nullptr;

        ListNode* ptrA = headA;
        ListNode* ptrB = headB;

        // When one pointer reaches end, switch to other list
        // This compensates for length difference
        while (ptrA != ptrB) {
            ptrA = ptrA ? ptrA->next : headB;
            ptrB = ptrB ? ptrB->next : headA;
        }

        return ptrA; // Either intersection point or nullptr
    }
};
```

Solution 3: Hash Set

Time Complexity: O(m + n)
Space Complexity: O(m) or O(n)

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        if (!headA || !headB) return nullptr;

        unordered_set<ListNode*> visited;
        ListNode* curr = headA;

        // Store all nodes from list A
        while (curr) {
            visited.insert(curr);
            curr = curr->next;
        }

        // Check list B for any visited node
        curr = headB;
        while (curr) {
            if (visited.find(curr) != visited.end()) {
                return curr;
            }
            curr = curr->next;
        }
        return nullptr;
    }
};
```

Solution 4: Marking Nodes (Modification)

Time Complexity: O(m + n)
Space Complexity: O(1) - but modifies the list

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        if (!headA || !headB) return nullptr;

        // Mark all nodes in list A
        ListNode* curr = headA;
        while (curr) {
            curr->val = -curr->val; // Mark by negating value
            curr = curr->next;
        }

        // Find first marked node in list B
        curr = headB;
        ListNode* intersection = nullptr;
        while (curr) {
            if (curr->val < 0) {
                intersection = curr;
                break;
            }
            curr = curr->next;
        }

        // Restore list A values
        curr = headA;
        while (curr) {
            curr->val = -curr->val;
            curr = curr->next;
        }
        return intersection;
    }
};
```

Solution 5: Using Cycle Detection with Dummy Node

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        if (!headA || !headB) return nullptr;

        // Create a cycle by connecting list A's tail to list B's head
        ListNode* tailA = headA;
        while (tailA->next) {
            tailA = tailA->next;
        }
        tailA->next = headB; // Create cycle

        // Use Floyd's cycle detection to find intersection
        ListNode* slow = headA;
        ListNode* fast = headA;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;

            if (slow == fast) {
                // Cycle detected, find intersection
                ListNode* ptr1 = headA;
                ListNode* ptr2 = slow;

                while (ptr1 != ptr2) {
                    ptr1 = ptr1->next;
                    ptr2 = ptr2->next;
                }
                // Restore the list
                tailA->next = nullptr;
                return ptr1;
            }
        }
        // No cycle, restore and return null
        tailA->next = nullptr;
        return nullptr;
    }
};
```

Solution 6: Reverse Both Lists

Time Complexity: O(m + n)
Space Complexity: O(1) - but modifies the lists

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        if (!headA || !headB) return nullptr;

        // Reverse both lists
        ListNode* reversedA = reverseList(headA);
        ListNode* reversedB = reverseList(headB);

        // If they intersect, the reversed lists will share
common prefix
        ListNode* intersection = nullptr;
        ListNode* ptrA = reversedA;
        ListNode* ptrB = reversedB;

        while (ptrA && ptrB && ptrA == ptrB) {
            intersection = ptrA;
            ptrA = ptrA->next;
            ptrB = ptrB->next;
        }

        // Restore original lists
        reverseList(reversedA);
        reverseList(reversedB);

        return intersection;
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

Solution 7: Using Difference in Lengths (Alternative)

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        if (!headA || !headB) return nullptr;

        ListNode* ptrA = headA;
        ListNode* ptrB = headB;

        // First pass: find if there's intersection and get
lengths
        int lenA = 1, lenB = 1;
        while (ptrA->next) {
            ptrA = ptrA->next;
            lenA++;
        }
        while (ptrB->next) {
            ptrB = ptrB->next;
            lenB++;
        }

        // If tails are different, no intersection
        if (ptrA != ptrB) return nullptr;

        // Reset pointers
        ptrA = headA;
        ptrB = headB;

        // Move longer list pointer forward by difference
        if (lenA > lenB) {
            for (int i = 0; i < lenA - lenB; i++) {
                ptrA = ptrA->next;
            }
        } else {
            for (int i = 0; i < lenB - lenA; i++) {
                ptrB = ptrB->next;
```

```
        }
    }

    // Find intersection point
    while (ptrA != ptrB) {
        ptrA = ptrA->next;
        ptrB = ptrB->next;
    }

    return ptrA;
  }
};
```

Solution 8: Recursive Approach

Time Complexity: O(m + n)
Space Complexity: O(m + n) - recursion stack

```
class Solution {
public:
  ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
      unordered_set<ListNode*> visited;
      return findIntersection(headA, headB, visited);
  }

private:
  ListNode* findIntersection(ListNode* nodeA, ListNode*
nodeB, unordered_set<ListNode*>& visited) {
      if (!nodeA && !nodeB) return nullptr;

      if (nodeA) {
          if (visited.find(nodeA) != visited.end()) {
              return nodeA;
          }
          visited.insert(nodeA);
      }

      if (nodeB) {
          if (visited.find(nodeB) != visited.end()) {
              return nodeB;
          }
          visited.insert(nodeB);
      }

      ListNode* nextA = nodeA ? nodeA->next : nullptr;
      ListNode* nextB = nodeB ? nodeB->next : nullptr;

      return findIntersection(nextA, nextB, visited);
  }
};
```

206 : Problem : Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:
Input: `head = [1,2,3,4,5]`
Output: `[5,4,3,2,1]`

Example 2:
Input: `head = [1,2]`
Output: `[2,1]`

Example 3:
Input: `head = []`
Output: `[]`

Solution 1: Iterative (Three Pointers)

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

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      ListNode* prev = nullptr;
      ListNode* curr = head;

      while (curr) {
          ListNode* next = curr->next;
          curr->next = prev;
          prev = curr;
          curr = next;
      }

      return prev;
  }
};
```

Solution 2: Recursive

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      // Base case
      if (!head || !head->next) return head;

      // Recursively reverse the rest of the list
      ListNode* newHead = reverseList(head->next);

      // Reverse the current connection
      head->next->next = head;
      head->next = nullptr;

      return newHead;
  }
};
```

Solution 3: Iterative with Dummy Node

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

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      ListNode* dummy = nullptr;
      ListNode* curr = head;

      while (curr) {
          ListNode* next = curr->next;
          curr->next = dummy;
          dummy = curr;
          curr = next;
      }
      return dummy;
  }
};
```

Solution 4: Stack-Based Approach

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

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      if (!head) return nullptr;

      stack<ListNode*> st;
      ListNode* curr = head;

      // Push all nodes to stack
      while (curr) {
          st.push(curr);
          curr = curr->next;
      }

      // Pop nodes to create reversed list
      ListNode* newHead = st.top();
      st.pop();
      curr = newHead;

      while (!st.empty()) {
          curr->next = st.top();
          st.pop();
          curr = curr->next;
      }

      curr->next = nullptr;
      return newHead;
  }
};
```

Solution 5: Recursive with Helper Function

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

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      return reverseHelper(head, nullptr);
  }

private:
  ListNode* reverseHelper(ListNode* curr, ListNode*
prev) {
      if (!curr) return prev;

      ListNode* next = curr->next;
      curr->next = prev;
      return reverseHelper(next, curr);
  }
};
```

Solution 6: In-Place Modification with Two Pointers

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

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      if (!head || !head->next) return head;

      ListNode* newHead = nullptr;
      ListNode* curr = head;

      while (curr) {
          ListNode* nextNode = curr->next;
          curr->next = newHead;
          newHead = curr;
          curr = nextNode;
      }
      return newHead;
  }
};
```

Solution 7: Using Vector/Array

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

```
class Solution {
public:
  ListNode* reverseList(ListNode* head) {
      if (!head) return nullptr;

      vector<ListNode*> nodes;
      ListNode* curr = head;

      // Store all nodes in vector
```

```
        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        // Reverse the connections
        for (int i = nodes.size() - 1; i > 0; i--) {
            nodes[i]->next = nodes[i - 1];
        }
        nodes[0]->next = nullptr;

        return nodes[nodes.size() - 1];
    }
};
```

Solution 8: Tail Recursive Optimization

Time Complexity: O(n)
Space Complexity: O(n) - but can be optimized by compiler

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        return reverse(head, nullptr);
    }
private:
    ListNode* reverse(ListNode* curr, ListNode* prev) {
        if (!curr) return prev;

        ListNode* next = curr->next;
        curr->next = prev;
        return reverse(next, curr);
    }
};
```

Solution 9: Iterative with Explicit Memory Management

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

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            // Store next pointer before modifying
            ListNode* nextTemp = curr->next;

            // Reverse the link
            curr->next = prev;

            // Move pointers forward
            prev = curr;
            curr = nextTemp;
        }
        return prev;
    }
};
```

Solution 10: Functional Style with Accumulator

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

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        return reverseAccum(head, nullptr);
    }

private:
    ListNode* reverseAccum(ListNode* head, ListNode*
accum) {
        if (!head) return accum;

        ListNode* nextHead = head->next;
        head->next = accum;
        return reverseAccum(nextHead, head);
    }
};
```

234 : Problem: **Palindrome Linked List**

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:
Input: `head = [1,2,2,1]`
Output: `true`

Example 2:
Input: `head = [1,2]`
Output: `false`

Example 3:
Input: `head = [1]`
Output: `true`

Follow up: Could you do it in O(n) time and O(1) space?

Solution 1: Reverse Second Half (Optimal)

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

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        // Step 1: Find the middle using slow and fast
pointers
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Step 2: Reverse the second half
        ListNode* secondHalf = reverseList(slow);
        ListNode* firstHalf = head;
        ListNode* secondHalfCopy = secondHalf;

        // Step 3: Compare both halves
        bool isPal = true;
        while (secondHalf) {
            if (firstHalf->val != secondHalf->val) {
                isPal = false;
                break;
            }
            firstHalf = firstHalf->next;
            secondHalf = secondHalf->next;
        }

        // Step 4: Restore the list (optional but good practice)
        reverseList(secondHalfCopy);

        return isPal;
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 2: Using Stack

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

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        stack<int> st;
        ListNode* curr = head;

        // Push all elements to stack
        while (curr) {
            st.push(curr->val);
            curr = curr->next;
        }

        // Compare with original list
        curr = head;
        while (curr) {
            if (curr->val != st.top()) {
                return false;
            }
            st.pop();
            curr = curr->next;
        }
        return true;
    }
};
```

Solution 3: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode* front = head;
        return checkPalindrome(head, front);
    }
private:
    bool checkPalindrome(ListNode* curr, ListNode*&
front) {
        if (!curr) return true;



     // Recursively go to the end
        if (!checkPalindrome(curr->next, front)) {
            return false;
        }

        // Compare current node with front
        if (curr->val != front->val) {
            return false;
        }

        front = front->next;
```

```
        return true;
    }
};
```

Solution 4: Copy to Array and Two Pointers

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

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        vector<int> values;
        ListNode* curr = head;

        // Copy linked list to array
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        // Check palindrome using two pointers
        int left = 0, right = values.size() - 1;
        while (left < right) {
            if (values[left] != values[right]) {
                return false;
            }
            left++;
            right--;
        }

        return true;
    }
};
```

Solution 5: Find Middle and Compare with Stack (Half Stack)

Time Complexity: O(n)
Space Complexity: O(n/2)

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        // Find middle using slow and fast pointers
        ListNode* slow = head;
        ListNode* fast = head;
        stack<int> st;

        // Push first half to stack
        while (fast && fast->next) {
            st.push(slow->val);
            slow = slow->next;
            fast = fast->next->next;
        }

        // If odd number of nodes, skip middle
        if (fast) {
            slow = slow->next;
        }

        // Compare second half with stack
        while (slow) {
            if (slow->val != st.top()) {
                return false;
            }
            st.pop();
            slow = slow->next;
        }

        return true;
    }
};
```

Solution 6: Reverse First Half and Compare

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

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;

        // Reverse first half while finding middle
        while (fast && fast->next) {
            fast = fast->next->next;

            // Reverse the slow pointer's path
            ListNode* next = slow->next;
            slow->next = prev;
            prev = slow;
            slow = next;
        }

        // Handle odd length
        ListNode* firstHalf = prev;
        ListNode* secondHalf = (fast) ? slow->next : slow;

        // Compare both halves
        bool result = true;
        ListNode* firstHalfCopy = firstHalf;
        ListNode* secondHalfCopy = secondHalf;

        while (firstHalf && secondHalf) {
            if (firstHalf->val != secondHalf->val) {
                result = false;
                break;
            }
            firstHalf = firstHalf->next;
            secondHalf = secondHalf->next;
        }
        // Restore the list
        while (firstHalfCopy) {
            ListNode* next = firstHalfCopy->next;
            firstHalfCopy->next = slow;
            slow = firstHalfCopy;
            firstHalfCopy = next;
        }
        return result;
    }
};
```

Solution 7: Using Deque

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

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        deque<int> dq;
        ListNode* curr = head;

        // Add all elements to deque
        while (curr) {
            dq.push_back(curr->val);
            curr = curr->next;
        }
        // Compare from both ends
        while (dq.size() > 1) {
            if (dq.front() != dq.back()) {
                return false;
            }
            dq.pop_front();
            dq.pop_back();
        }
        return true;
    }
};
```

Solution 8: Two Pass with Length Calculation

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

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        // Calculate length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Find middle node
        ListNode* slow = head;
        for (int i = 0; i < length / 2; i++) {
            slow = slow->next;
        }
        // Reverse second half
        ListNode* secondHalf = reverseList(slow);
        ListNode* firstHalf = head;

        // Compare both halves
        for (int i = 0; i < length / 2; i++) {
            if (firstHalf->val != secondHalf->val) {
                reverseList(secondHalf); // Restore
                return false;
            }
            firstHalf = firstHalf->next;
            secondHalf = secondHalf->next;
        }
        reverseList(slow); // Restore original list
        return true;
    }
private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

Solution 9: Hash-Based Approach (Alternative)

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

```cpp
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if (!head || !head->next) return true;

        string forward = "";
        string backward = "";
        ListNode* curr = head;

        // Build both strings
        while (curr) {
            forward += to_string(curr->val);
            backward = to_string(curr->val) + backward;
            curr = curr->next;
        }
        return forward == backward;
    }
};
```

Solution 10: Optimized Recursive with Global Pointer

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

```cpp
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode* globalHead = head;
        return recursiveCheck(head, globalHead);
    }
private:
    bool recursiveCheck(ListNode* node, ListNode*&
globalHead) {
        if (!node) return true;

        bool restIsPalindrome = recursiveCheck(node->next,
globalHead);

        if (!restIsPalindrome) return false;
        if (node->val != globalHead->val) return false;

        globalHead = globalHead->next;
        return true;
    }
};
```

328 : Problem : Odd Even Linked List

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The first node is considered odd, and the second node is even, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

Example 1:
Input: `head = [1,2,3,4,5]`
Output: `[1,3,5,2,4]`

Example 2:
Input: `head = [2,1,3,5,6,4,7]`
Output: `[2,3,6,7,1,5,4]`

Example 3:
Input: `head = [1]`
Output: `[1]`

Solution 1: Separate Odd and Even Lists

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

```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* odd = head;
        ListNode* even = head->next;
        ListNode* evenHead = even;

        while (even && even->next) {
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }

        odd->next = evenHead;
        return head;
    }
};
```

Solution 2: Two Pointer Approach

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

```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* oddTail = head;
        ListNode* evenTail = head->next;
        ListNode* evenHead = evenTail;

        while (evenTail && evenTail->next) {
            // Connect odd to next odd
            oddTail->next = evenTail->next;
            oddTail = oddTail->next;

            // Connect even to next even
            evenTail->next = oddTail->next;
            evenTail = evenTail->next;
        }

        // Connect odd list to even list
        oddTail->next = evenHead;
        return head;
    }
};
```

Solution 3: Using Dummy Nodes

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

```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* oddDummy = new ListNode(0);
        ListNode* evenDummy = new ListNode(0);
        ListNode* oddTail = oddDummy;
        ListNode* evenTail = evenDummy;

        ListNode* curr = head;
        int index = 1;

        while (curr) {
            if (index % 2 == 1) { // Odd index
                oddTail->next = curr;
                oddTail = oddTail->next;
            } else { // Even index
                evenTail->next = curr;
                evenTail = evenTail->next;
            }
            curr = curr->next;
            index++;
        }

        // Connect odd list to even list
        oddTail->next = evenDummy->next;
        evenTail->next = nullptr;

        ListNode* result = oddDummy->next;
        delete oddDummy;
        delete evenDummy;
        return result;
    }
};
```

Solution 4: In-Place Rearrangement

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

```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* odd = head;
        ListNode* even = head->next;

        while (even && even->next) {
            // Remove the next odd node from its position
            ListNode* nextOdd = even->next;
            even->next = nextOdd->next;

            // Insert the odd node after current odd
            nextOdd->next = odd->next;
            odd->next = nextOdd;

            // Move pointers forward
            odd = odd->next;
            even = even->next;
        }
        return head;
    }
};
```

Solution 5: Vector/Array Approach

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

```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head) return head;

        vector<ListNode*> nodes;
        ListNode* curr = head;

        // Store all nodes in vector
        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        // Rebuild list with odd indices first, then even indices
        ListNode* dummy = new ListNode(0);
        curr = dummy;

        // Add odd indices (1-based: 1, 3, 5, ...)
        for (int i = 0; i < nodes.size(); i += 2) {
            curr->next = nodes[i];
            curr = curr->next;
```

```
        }

        // Add even indices (1-based: 2, 4, 6, ...)
        for (int i = 1; i < nodes.size(); i += 2) {
            curr->next = nodes[i];
            curr = curr->next;
        }
        curr->next = nullptr;
        ListNode* result = dummy->next;
        delete dummy;
        return result;
    }
};
```

Solution 6: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* oddHead = head;
        ListNode* evenHead = head->next;

        rearrange(oddHead, evenHead);

        // Find the end of odd list and connect to even head
        ListNode* curr = oddHead;
        while (curr->next) {
            curr = curr->next;
        }
        curr->next = evenHead;
        return oddHead;
    }

private:
    void rearrange(ListNode* odd, ListNode* even) {
        if (!even || !even->next) {
            if (odd) odd->next = nullptr;
            if (even) even->next = nullptr;
            return;
        }
        odd->next = even->next;
        even->next = even->next->next;

        rearrange(odd->next, even->next);
    }
};
```

Solution 7: Iterative with Explicit Index Tracking

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

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* odd = head;
        ListNode* even = head->next;
        ListNode* evenStart = even;

        ListNode* curr = even->next;
        bool isOdd = true;

        while (curr) {
            if (isOdd) {
                odd->next = curr;
                odd = odd->next;
            } else {
                even->next = curr;
                even = even->next;
            }
            curr = curr->next;
            isOdd = !isOdd;
        }
        // Connect odd list to even list and terminate even list
        odd->next = evenStart;
        even->next = nullptr;
        return head;
    }
};
```

Solution 8: Using Two Separate Lists

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

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* oddHead = nullptr, *oddTail = nullptr;
        ListNode* evenHead = nullptr, *evenTail = nullptr;
        ListNode* curr = head;
        int index = 1;

        while (curr) {
            if (index % 2 == 1) { // Odd position
                if (!oddHead) {
                    oddHead = curr;
                    oddTail = curr;
                } else {
                    oddTail->next = curr;
                    oddTail = oddTail->next;
                }
            } else { // Even position
                if (!evenHead) {
                    evenHead = curr;
                    evenTail = curr;
                } else {
                    evenTail->next = curr;
                    evenTail = evenTail->next;
                }
            }
            curr = curr->next;
            index++;
        }

        // Connect odd list to even list
        if (oddTail) oddTail->next = evenHead;
        if (evenTail) evenTail->next = nullptr;

        return oddHead;
    }
};
```

Solution 9: Modified In-Place Approach

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

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* lastOdd = head;
        ListNode* firstEven = head->next;
        ListNode* curr = firstEven;

        while (curr && curr->next) {
            // Move next odd node to after lastOdd
            ListNode* nextOdd = curr->next;
            curr->next = nextOdd->next;
            nextOdd->next = firstEven;
            lastOdd->next = nextOdd;

            // Update pointers
            lastOdd = nextOdd;
            curr = curr->next;
        }
        return head;
    }
};
```

Solution 10: Step-by-Step Pointer Manipulation

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

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (!head || !head->next || !head->next->next) return
head;

        ListNode* odd = head;
        ListNode* even = head->next;

        while (even && even->next) {
            // Store the next odd node
            ListNode* temp = even->next;

            // Remove the odd node from even section
            even->next = temp->next;

            // Insert the odd node between current odd and
even head
            temp->next = odd->next;
            odd->next = temp;

            // Move pointers
            odd = odd->next;
            even = even->next;
        }

        return head;
    }
};
```

355. Problem: **Design Twitter**

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user, and see the 10 most recent tweets in the user's news feed.

Requirements:
- postTweet(userId, tweetId): Composes a new tweet with ID tweetId by the user userId.
- getNewsFeed(userId): Retrieves the 10 most recent tweet IDs in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user themselves. Tweets must be ordered from most recent to least recent.
- follow(followerId, followeeId): The user with ID followerId started following the user with ID followeeId.
- unfollow(followerId, followeeId): The user with ID followerId started unfollowing the user with ID followeeId.

Example:

```
Twitter twitter = new Twitter();
twitter.postTweet(1, 5); // User 1 posts a new tweet (id =
5).
twitter.getNewsFeed(1);  // User 1's news feed should
return a list with 1 tweet id -> [5].
twitter.follow(1, 2);    // User 1 follows user 2.
twitter.postTweet(2, 6); // User 2 posts a new tweet (id =
6).
twitter.getNewsFeed(1);  // User 1's news feed should
return a list with 2 tweet ids -> [6, 5].
```

```
twitter.unfollow(1, 2);  // User 1 unfollows user 2.
twitter.getNewsFeed(1);  // User 1's news feed should return a list with 1 tweet id -> [5].
```

Solution 1: Object-Oriented Design with Priority Queue

Time Complexity:
- postTweet: O(1)
- getNewsFeed: O(n log k) where n is total tweets from all followed users
- follow: O(1)
- unfollow: O(1)

Space Complexity: O(U + T) where U is users and T is tweets

```
class Twitter {
private:
    struct Tweet {
        int tweetId;
        int timestamp;
        Tweet* next;
        Tweet(int id, int time) : tweetId(id), timestamp(time), next(nullptr) {}
    };

    struct User {
        int userId;
        unordered_set<int> following;
        Tweet* tweetHead;
        User(int id) : userId(id), tweetHead(nullptr) {
            follow(id); // User follows themselves
        }

        void follow(int id) {
            following.insert(id);
        }

        void unfollow(int id) {
            if (id != userId) { // Cannot unfollow yourself
                following.erase(id);
            }
        }

        void post(int id, int time) {
            Tweet* newTweet = new Tweet(id, time);
            newTweet->next = tweetHead;
            tweetHead = newTweet;
        }
    };

    int time;
    unordered_map<int, User*> userMap;

    User* getUser(int userId) {
        if (userMap.find(userId) == userMap.end()) {
            userMap[userId] = new User(userId);
        }
        return userMap[userId];
    }

public:
    Twitter() {
        time = 0;
    }

    void postTweet(int userId, int tweetId) {
        User* user = getUser(userId);
        user->post(tweetId, time++);
    }

    vector<int> getNewsFeed(int userId) {
        vector<int> newsFeed;
        if (userMap.find(userId) == userMap.end()) return newsFeed;

        User* user = userMap[userId];

        // Max heap based on timestamp
        auto comp = [](const Tweet* a, const Tweet* b) {
            return a->timestamp < b->timestamp;
        };
        priority_queue<Tweet*, vector<Tweet*>, decltype(comp)> pq(comp);

        // Add latest tweet from each followed user
        for (int followeeId : user->following) {
            if (userMap.find(followeeId) != userMap.end() && userMap[followeeId]->tweetHead) {
                pq.push(userMap[followeeId]->tweetHead);
            }
        }

        // Get 10 most recent tweets
        int count = 0;
        while (!pq.empty() && count < 10) {
            Tweet* tweet = pq.top();
            pq.pop();
            newsFeed.push_back(tweet->tweetId);

            // Add next tweet from the same user
            if (tweet->next) {
                pq.push(tweet->next);
            }

            count++;
        }

        return newsFeed;
    }

    void follow(int followerId, int followeeId) {
        User* follower = getUser(followerId);
        follower->follow(followeeId);
    }

    void unfollow(int followerId, int followeeId) {
        if (userMap.find(followerId) != userMap.end()) {
            userMap[followerId]->unfollow(followeeId);
        }
    }
};
```

Solution 2: Simplified HashMap Approach

Time Complexity:
- postTweet: O(1)
- getNewsFeed: O(n log 10) where n is total tweets
- follow: O(1)
- unfollow: O(1)

Space Complexity: O(U + T)

```
class Twitter {
private:
    int time;
    unordered_map<int, vector<pair<int, int>>> userTweets; // userId -> [(timestamp, tweetId)]
    unordered_map<int, unordered_set<int>> following; // userId -> set of followeeIds

public:
    Twitter() {
        time = 0;
    }

    void postTweet(int userId, int tweetId) {
        userTweets[userId].push_back({time++, tweetId});
    }

    vector<int> getNewsFeed(int userId) {
        // User follows themselves
        following[userId].insert(userId);

        // Min heap to keep top 10 tweets
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> minHeap;

        for (int followeeId : following[userId]) {
            if (userTweets.find(followeeId) != userTweets.end()) {
                const vector<pair<int, int>>& tweets = userTweets[followeeId];
                for (int i = tweets.size() - 1; i >= max(0, (int)tweets.size() - 10); i--) {
                    if (minHeap.size() < 10) {
                        minHeap.push(tweets[i]);
                    } else if (tweets[i].first > minHeap.top().first) {
                        minHeap.pop();
                        minHeap.push(tweets[i]);
                    } else {
                        break; // Early termination since tweets are sorted by time
                    }
                }
            }
        }

        // Convert min heap to result (most recent first)
        vector<int> result(minHeap.size());
        for (int i = minHeap.size() - 1; i >= 0; i--) {
            result[i] = minHeap.top().second;
            minHeap.pop();
        }
        return result;
    }

    void follow(int followerId, int followeeId) {
        following[followerId].insert(followeeId);
    }

    void unfollow(int followerId, int followeeId) {
        if (following[followerId].count(followeeId)) {
            following[followerId].erase(followeeId);
        }
    }
};
```

Solution 3: Using Linked List for Tweets with Merge K Sorted Lists

Time Complexity:
- postTweet: O(1)
- getNewsFeed: O(10 * k) where k is number of followed users
- follow: O(1)
- unfollow: O(1)

Space Complexity: O(U + T)

```
class Twitter {
private:
    struct Tweet {
        int id;
        int time;
        Tweet* next;
        Tweet(int id, int time) : id(id), time(time), next(nullptr) {}
    };

    int globalTime;
    unordered_map<int, Tweet*> userTweets;
    unordered_map<int, unordered_set<int>> userFollowing;

public:
    Twitter() : globalTime(0) {}

    void postTweet(int userId, int tweetId) {
```

```
        Tweet* newTweet = new Tweet(tweetId,
globalTime++);
        newTweet->next = userTweets[userId];
        userTweets[userId] = newTweet;
    }

    vector<int> getNewsFeed(int userId) {
        // User follows themselves
        userFollowing[userId].insert(userId);

        // Custom comparator for max heap
        auto comp = [](Tweet* a, Tweet* b) { return a->time <
b->time; };
        priority_queue<Tweet*, vector<Tweet*>,
decltype(comp)> pq(comp);

        // Add latest tweet from each followed user
        for (int followeeId : userFollowing[userId]) {
            if (userTweets.count(followeeId) &&
userTweets[followeeId]) {
                pq.push(userTweets[followeeId]);
            }
        }
        vector<int> result;
        for (int i = 0; i < 10 && !pq.empty(); i++) {
            Tweet* tweet = pq.top();
            pq.pop();
            result.push_back(tweet->id);

            // Add next tweet from same user
            if (tweet->next) {
                pq.push(tweet->next);
            }
        }
        return result;
    }
    void follow(int followerId, int followeeId) {
        userFollowing[followerId].insert(followeeId);
    }

    void unfollow(int followerId, int followeeId) {
        if (userFollowing[followerId].count(followeeId)) {
            userFollowing[followerId].erase(followeeId);
        }
    }
};
```

Solution 4: Using Vector and Sorting

Time Complexity:
- postTweet: O(1)
- getNewsFeed: O(n log n) where n is total tweets
- follow: O(1)
- unfollow: O(1)

Space Complexity: O(U + T)

```
class Twitter {
private:
    int time;
    unordered_map<int, vector<pair<int, int>>> tweets; //
userId -> [(time, tweetId)]
    unordered_map<int, unordered_set<int>> follows; //
userId -> set of followeeIds

public:
    Twitter() : time(0) {}

    void postTweet(int userId, int tweetId) {
        tweets[userId].emplace_back(time++, tweetId);
    }

    vector<int> getNewsFeed(int userId) {
        follows[userId].insert(userId); // User follows
themselves

        vector<pair<int, int>> allTweets;

        // Collect all tweets from followed users
        for (int followeeId : follows[userId]) {
            if (tweets.count(followeeId)) {
                allTweets.insert(allTweets.end(),
                               tweets[followeeId].begin(),
                               tweets[followeeId].end());
            }
        }

        // Sort by time (most recent first)
        sort(allTweets.rbegin(), allTweets.rend());

        // Get top 10
        vector<int> result;
        for (int i = 0; i < min(10, (int)allTweets.size()); i++) {
            result.push_back(allTweets[i].second);
        }

        return result;
    }

    void follow(int followerId, int followeeId) {
        follows[followerId].insert(followeeId);
    }

    void unfollow(int followerId, int followeeId) {
        if (follows[followerId].count(followeeId)) {
            follows[followerId].erase(followeeId);
        }
    }
};
```

Solution 5: Optimized with Limited Tweet Storage

Time Complexity:
- postTweet: O(1)
- getNewsFeed: O(10 * k)
- follow: O(1)
- unfollow: O(1)

Space Complexity: O(U + T) but limits tweet storage per user

```
class Twitter {
private:
    struct Tweet {
        int id;
        int time;
        Tweet(int id, int time) : id(id), time(time) {}
    };

    int globalTime;
    const int FEED_SIZE = 10;
    unordered_map<int, deque<Tweet>> userTweets; //
Store only recent tweets
    unordered_map<int, unordered_set<int>> following;

public:
    Twitter() : globalTime(0) {}

    void postTweet(int userId, int tweetId) {
        userTweets[userId].push_front(Tweet(tweetId,
globalTime++));

        // Keep only recent tweets to save space
        if (userTweets[userId].size() > FEED_SIZE) {
            userTweets[userId].pop_back();
        }
    }

    vector<int> getNewsFeed(int userId) {
        following[userId].insert(userId); // Follow self

        // Min heap to get top 10 tweets
        auto comp = [](const Tweet& a, const Tweet& b) {
            return a.time > b.time;
        };
        priority_queue<Tweet, vector<Tweet>,
decltype(comp)> pq(comp);

        for (int followeeId : following[userId]) {
            if (userTweets.count(followeeId)) {
                for (const Tweet& tweet :
userTweets[followeeId]) {
                    if (pq.size() < FEED_SIZE) {
                        pq.push(tweet);
                    } else if (tweet.time > pq.top().time) {
                        pq.pop();
                        pq.push(tweet);
                    }
                }
            }
        }

        vector<int> result(pq.size());
        for (int i = pq.size() - 1; i >= 0; i--) {
            result[i] = pq.top().id;
            pq.pop();
        }

        return result;
    }

    void follow(int followerId, int followeeId) {
        following[followerId].insert(followeeId);
    }

    void unfollow(int followerId, int followeeId) {
        if (following[followerId].count(followeeId)) {
            following[followerId].erase(followeeId);
        }
    }
};
```

Solution 6: Using Multiset for Automatic Sorting

Time Complexity:
- postTweet: O(log n)
- getNewsFeed: O(10)
- follow: O(1)
- unfollow: O(1)

Space Complexity: O(U + T)

```
class Twitter {
private:
    int time;
    unordered_map<int, unordered_set<int>> following;
    multiset<pair<int, pair<int, int>>, greater<pair<int,
int>>> allTweets; // {time, {userId, tweetId}}

public:
    Twitter() : time(0) {}

    void postTweet(int userId, int tweetId) {
        allTweets.insert({time++, {userId, tweetId}});
    }

    vector<int> getNewsFeed(int userId) {
        following[userId].insert(userId); // Follow self

        vector<int> result;
        int count = 0;

        for (const auto& tweet : allTweets) {
            int tweetUserId = tweet.second.first;
```

```
        int tweetId = tweet.second.second;

        if (following[userId].count(tweetUserId)) {
            result.push_back(tweetId);
            count++;
            if (count >= 10) break;
        }
    }
    return result;
}

void follow(int followerId, int followeeId) {
    following[followerId].insert(followeeId);
}

void unfollow(int followerId, int followeeId) {
    if (following[followerId].count(followeeId)) {
        following[followerId].erase(followeeId);
    }
}
};
```

445. Problem: **Add Two Numbers II**

You are given two non-empty linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:
Input: `l1 = [7,2,4,3], l2 = [5,6,4]`
Output: `[7,8,0,7]`
Explanation: `7243 + 564 = 7807`

Example 2:
Input: `l1 = [2,4,3], l2 = [5,6,4]`
Output: `[8,0,7]`
Explanation: `243 + 564 = 807`

Example 3:
Input: `l1 = [0], l2 = [0]`
Output: `[0]`

Solution 1: Using Stacks (Most Intuitive)

Time Complexity: O(m + n)
Space Complexity: O(m + n)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        stack<int> s1, s2;

        // Push all digits to stacks
        ListNode* curr1 = l1;
        while (curr1) {
            s1.push(curr1->val);
            curr1 = curr1->next;
        }

        ListNode* curr2 = l2;
        while (curr2) {
            s2.push(curr2->val);
            curr2 = curr2->next;
        }

        ListNode* result = nullptr;
        int carry = 0;

        // Add digits from stacks
        while (!s1.empty() || !s2.empty() || carry) {
            int sum = carry;

            if (!s1.empty()) {
                sum += s1.top();
                s1.pop();
            }
            if (!s2.empty()) {
                sum += s2.top();
                s2.pop();
            }

            carry = sum / 10;
            ListNode* newNode = new ListNode(sum % 10);
            newNode->next = result;
            result = newNode;
        }
        return result;
    }
};
```

Solution 2: Reverse Lists, Add, Reverse Back

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        // Reverse both lists
        l1 = reverseList(l1);
        l2 = reverseList(l2);

        // Add the reversed lists
        ListNode* result = addLists(l1, l2);

        // Reverse the result
        return reverseList(result);
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }

    ListNode* addLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;
        int carry = 0;

        while (l1 || l2 || carry) {
            int sum = carry;
            if (l1) {
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2) {
                sum += l2->val;
                l2 = l2->next;
            }

            carry = sum / 10;
            curr->next = new ListNode(sum % 10);
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 3: Recursive Approach with Length Calculation

Time Complexity: O(m + n)
Space Complexity: O(max(m, n)) - recursion stack

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        int len1 = getLength(l1);
        int len2 = getLength(l2);

        // Pad the shorter list with zeros
        if (len1 < len2) {
            l1 = padList(l1, len2 - len1);
        } else if (len2 < len1) {
            l2 = padList(l2, len1 - len2);
        }

        // Add lists recursively
        int carry = 0;
        ListNode* result = addListsRecursive(l1, l2, carry);

        // Handle final carry
        if (carry) {
            ListNode* newHead = new ListNode(carry);
            newHead->next = result;
            return newHead;
        }

        return result;
    }

private:
    int getLength(ListNode* head) {
        int length = 0;
        while (head) {
            length++;
            head = head->next;
        }
        return length;
    }

    ListNode* padList(ListNode* head, int padding) {
        while (padding--) {
            ListNode* newNode = new ListNode(0);
            newNode->next = head;
            head = newNode;
        }
        return head;
    }

    ListNode* addListsRecursive(ListNode* l1, ListNode*
l2, int& carry) {
        if (!l1 && !l2) return nullptr;

        // Recursively process next nodes
        ListNode* nextNode = addListsRecursive(l1->next,
l2->next, carry);

        // Process current nodes
        int sum = l1->val + l2->val + carry;
        carry = sum / 10;

        ListNode* currentNode = new ListNode(sum % 10);
        currentNode->next = nextNode;

        return currentNode;
    }
};
```

Solution 4: Using Vectors

Time Complexity: O(m + n)
Space Complexity: O(m + n)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        vector<int> nums1, nums2;

        // Store digits in vectors
        ListNode* curr = l1;
        while (curr) {
            nums1.push_back(curr->val);
            curr = curr->next;
        }

        curr = l2;
        while (curr) {
            nums2.push_back(curr->val);
            curr = curr->next;
        }

        // Add from end to beginning
        int i = nums1.size() - 1, j = nums2.size() - 1;
        int carry = 0;
        ListNode* result = nullptr;

        while (i >= 0 || j >= 0 || carry) {
            int sum = carry;
            if (i >= 0) sum += nums1[i--];
            if (j >= 0) sum += nums2[j--];

            carry = sum / 10;
            ListNode* newNode = new ListNode(sum % 10);
            newNode->next = result;
            result = newNode;
        }

        return result;
    }
};
```

Solution 5: In-place Modification (Longer List)

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        int len1 = getLength(l1);
        int len2 = getLength(l2);

        // Use the longer list for result
        ListNode* longer = (len1 >= len2) ? l1 : l2;
        ListNode* shorter = (len1 >= len2) ? l2 : l1;

        // Pad shorter list if needed
        int diff = abs(len1 - len2);
        shorter = padList(shorter, diff);

        // Add lists and store result in longer list
        int carry = addListsInPlace(longer, shorter);

        // Handle final carry
        if (carry) {
            ListNode* newHead = new ListNode(carry);
            newHead->next = longer;
            return newHead;
        }

        return longer;
    }

private:
    int getLength(ListNode* head) {
        int length = 0;
        while (head) {
            length++;
            head = head->next;
        }
        return length;
    }

    ListNode* padList(ListNode* head, int padding) {
        while (padding--) {
            ListNode* newNode = new ListNode(0);
            newNode->next = head;
            head = newNode;
        }
        return head;
    }

    int addListsInPlace(ListNode* l1, ListNode* l2) {
        if (!l1 && !l2) return 0;

        int carry = addListsInPlace(l1->next, l2->next);

        int sum = l1->val + l2->val + carry;
        l1->val = sum % 10;

        return sum / 10;
    }
};
```

Solution 6: Iterative with Two Passes

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        // Calculate lengths
        int len1 = 0, len2 = 0;
        ListNode* curr1 = l1, *curr2 = l2;

        while (curr1) { len1++; curr1 = curr1->next; }
        while (curr2) { len2++; curr2 = curr2->next; }

        // Find longer and shorter lists
        ListNode* longer = (len1 >= len2) ? l1 : l2;
        ListNode* shorter = (len1 >= len2) ? l2 : l1;

        // Add digits and store carry in a separate way
        ListNode* result = nullptr;
        int diff = abs(len1 - len2);

        // Process the extra part of longer list
        curr1 = longer;
        for (int i = 0; i < diff; i++) {
            ListNode* newNode = new ListNode(curr1->val);
            newNode->next = result;
            result = newNode;
            curr1 = curr1->next;
        }

        // Process both lists together
        curr2 = shorter;
        while (curr1) {
            int sum = curr1->val + curr2->val;
            ListNode* newNode = new ListNode(sum);
            newNode->next = result;
            result = newNode;
            curr1 = curr1->next;
            curr2 = curr2->next;
        }

        // Process carry through the result list
        ListNode* finalResult = nullptr;
        int carry = 0;
        while (result) {
            int sum = result->val + carry;
            carry = sum / 10;
            ListNode* newNode = new ListNode(sum % 10);
            newNode->next = finalResult;
            finalResult = newNode;
            result = result->next;
        }

        if (carry) {
            ListNode* newNode = new ListNode(carry);
            newNode->next = finalResult;
            finalResult = newNode;
        }

        return finalResult;
    }
};
```

Solution 7: Using Deque

Time Complexity: O(m + n)
Space Complexity: O(m + n)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2)
{
        deque<int> dq1, dq2;

        // Store digits in deques
        ListNode* curr = l1;
        while (curr) {
            dq1.push_back(curr->val);
            curr = curr->next;
        }

        curr = l2;
        while (curr) {
            dq2.push_back(curr->val);
            curr = curr->next;
        }

        // Add from back to front
        int carry = 0;
        ListNode* result = nullptr;

        while (!dq1.empty() || !dq2.empty() || carry) {
            int sum = carry;

            if (!dq1.empty()) {
                sum += dq1.back();
                dq1.pop_back();
            }
            if (!dq2.empty()) {
                sum += dq2.back();
                dq2.pop_back();
            }

            carry = sum / 10;
            ListNode* newNode = new ListNode(sum % 10);
            newNode->next = result;
            result = newNode;
        }

        return result;
    }
};
```

Solution 8: Convert to Numbers (Limited Use - May Overflow)

Time Complexity: O(m + n)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        long long num1 = 0, num2 = 0;

        // Convert lists to numbers
        ListNode* curr = l1;
        while (curr) {
            num1 = num1 * 10 + curr->val;
            curr = curr->next;
        }

        curr = l2;
        while (curr) {
            num2 = num2 * 10 + curr->val;
            curr = curr->next;
        }

        // Calculate sum
        long long sum = num1 + num2;

        // Convert sum back to linked list
        if (sum == 0) return new ListNode(0);

        ListNode* result = nullptr;
        while (sum > 0) {
            ListNode* newNode = new ListNode(sum % 10);
            newNode->next = result;
            result = newNode;
            sum /= 10;
        }

        return result;
    }
};
```

622. Problem: **Design Circular Queue**

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

Operations:
- `MyCircularQueue(k)`: Constructor, set the size of the queue to be k.
- `Front()`: Get the front item from the queue. If the queue is empty, return -1.
- `Rear()`: Get the last item from the queue. If the queue is empty, return -1.
- `enQueue(value)`: Insert an element into the circular queue. Return true if the operation is successful.
- `deQueue()`: Delete an element from the circular queue. Return true if the operation is successful.
- `isEmpty()`: Checks whether the circular queue is empty.
- `isFull()`: Checks whether the circular queue is full.

Example:

```
MyCircularQueue circularQueue = new MyCircularQueue(3); // set size to be 3
circularQueue.enQueue(1);  // return true
circularQueue.enQueue(2);  // return true
circularQueue.enQueue(3);  // return true
circularQueue.enQueue(4);  // return false (queue is full)
circularQueue.Rear();      // return 3
circularQueue.isFull();    // return true
circularQueue.deQueue();   // return true
circularQueue.enQueue(4);  // return true
circularQueue.Rear();      // return 4
```

Solution 1: Array Implementation with Two Pointers

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularQueue {
private:
    vector<int> data;
    int front;
    int rear;
    int size;
    int capacity;

public:
    MyCircularQueue(int k) {
        data.resize(k);
        front = 0;
        rear = -1;
        size = 0;
        capacity = k;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        rear = (rear + 1) % capacity;
        data[rear] = value;
        size++;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;
        front = (front + 1) % capacity;
        size--;
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return data[front];
    }

    int Rear() {
        if (isEmpty()) return -1;
        return data[rear];
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
    }
};
```

Solution 2: Array Implementation with Count

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularQueue {
private:
    vector<int> data;
    int head;
    int count;
    int capacity;

public:
    MyCircularQueue(int k) {
        data.resize(k);
        head = 0;
        count = 0;
        capacity = k;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        int tail = (head + count) % capacity;
        data[tail] = value;
        count++;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;
        head = (head + 1) % capacity;
        count--;
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return data[head];
    }

    int Rear() {
        if (isEmpty()) return -1;
        int tail = (head + count - 1) % capacity;
        return data[tail];
    }

    bool isEmpty() {
        return count == 0;
    }

    bool isFull() {
        return count == capacity;
    }
};
```

Solution 3: Linked List Implementation

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class Node {
public:
    int val;
    Node* next;
    Node(int value) : val(value), next(nullptr) {}
};

class MyCircularQueue {
private:
    Node* head;
    Node* tail;
    int size;
    int capacity;

public:
    MyCircularQueue(int k) {
        head = nullptr;
        tail = nullptr;
        size = 0;
        capacity = k;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        Node* newNode = new Node(value);
        if (isEmpty()) {
```

```
        head = tail = newNode;
    } else {
        tail->next = newNode;
        tail = newNode;
    }
    tail->next = head; // Make it circular
    size++;
    return true;
}

bool deQueue() {
    if (isEmpty()) return false;

    if (head == tail) { // Only one element
        delete head;
        head = tail = nullptr;
    } else {
        Node* temp = head;
        head = head->next;
        tail->next = head; // Update circular reference
        delete temp;
    }
    size--;
    return true;
}

int Front() {
    if (isEmpty()) return -1;
    return head->val;
}

int Rear() {
    if (isEmpty()) return -1;
    return tail->val;
}

bool isEmpty() {
    return size == 0;
}

bool isFull() {
    return size == capacity;
}

~MyCircularQueue() {
    while (!isEmpty()) {
        deQueue();
    }
}
};
```

Solution 4: Array with Flag Approach

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularQueue {
private:
    vector<int> data;
    int front;
    int rear;
    int capacity;
    bool full;

public:
    MyCircularQueue(int k) {
        data.resize(k);
        front = 0;
        rear = 0;
        capacity = k;
        full = false;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        data[rear] = value;
        rear = (rear + 1) % capacity;
        full = (rear == front);
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;

        front = (front + 1) % capacity;
        full = false;
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return data[front];
    }

    int Rear() {
        if (isEmpty()) return -1;
        int lastIndex = (rear - 1 + capacity) % capacity;
        return data[lastIndex];
    }

    bool isEmpty() {
        return (front == rear) && !full;
    }

    bool isFull() {
        return full;
    }
};
```

Solution 5: Doubly Linked List Implementation

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class Node {
public:
    int val;
    Node* prev;
    Node* next;
    Node(int value) : val(value), prev(nullptr), next(nullptr)
{}
};

class MyCircularQueue {
private:
    Node* head;
    Node* tail;
    int size;
    int capacity;

public:
    MyCircularQueue(int k) {
        head = nullptr;
        tail = nullptr;
        size = 0;
        capacity = k;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        Node* newNode = new Node(value);
        if (isEmpty()) {
            head = tail = newNode;
            head->next = head;
            head->prev = head;
        } else {
            newNode->prev = tail;
            newNode->next = head;
            tail->next = newNode;
            head->prev = newNode;
            tail = newNode;
        }
        size++;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;

        if (size == 1) {
            delete head;
            head = tail = nullptr;
        } else {
            Node* temp = head;
            head = head->next;
            head->prev = tail;
            tail->next = head;
            delete temp;
        }
        size--;
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return head->val;
    }

    int Rear() {
        if (isEmpty()) return -1;
        return tail->val;
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
    }

    ~MyCircularQueue() {
        while (!isEmpty()) {
            deQueue();
        }
    }
};
```

Solution 6: Vector with Modular Arithmetic

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularQueue {
private:
    vector<int> queue;
    int front;
    int rear;
    int maxSize;

public:
    MyCircularQueue(int k) {
        queue.resize(k);
        front = 0;
        rear = -1;
        maxSize = k;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        rear = (rear + 1) % maxSize;
```

```
        queue[rear] = value;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;

        if (front == rear) { // Only one element
            front = 0;
            rear = -1;
        } else {
            front = (front + 1) % maxSize;
        }
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return queue[front];
    }

    int Rear() {
        if (isEmpty()) return -1;
        return queue[rear];
    }

    bool isEmpty() {
        return rear == -1;
    }

    bool isFull() {
        return !isEmpty() && (rear + 1) % maxSize == front;
    }
};
```

Solution 7: Using Queue STL with Size Limit

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularQueue {
private:
    queue<int> q;
    int maxSize;

public:
    MyCircularQueue(int k) {
        maxSize = k;
    }

    bool enQueue(int value) {
        if (isFull()) return false;
        q.push(value);
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;
        q.pop();
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return q.front();
    }

    int Rear() {
        if (isEmpty()) return -1;
        return q.back();
    }

    bool isEmpty() {
        return q.empty();
    }

    bool isFull() {
        return q.size() == maxSize;
    }
};
```

Solution 8: Array with Wasted Space

Time Complexity: O(1) for all operations
Space Complexity: O(k+1)

```
class MyCircularQueue {
private:
    vector<int> data;
    int front;
    int rear;
    int capacity;

public:
    MyCircularQueue(int k) {
        data.resize(k + 1); // One extra space to distinguish
full vs empty
        front = 0;
        rear = 0;
        capacity = k + 1;
    }

    bool enQueue(int value) {
        if (isFull()) return false;

        data[rear] = value;
        rear = (rear + 1) % capacity;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;
        front = (front + 1) % capacity;
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return data[front];
    }

    int Rear() {
        if (isEmpty()) return -1;
        return data[(rear - 1 + capacity) % capacity];
    }

    bool isEmpty() {
        return front == rear;
    }

    bool isFull() {
        return (rear + 1) % capacity == front;
    }
};
```

641. Problem: **Design Circular Deque**

Design your implementation of the circular double-ended queue (deque).

Implement the `MyCircularDeque` class:
- `MyCircularDeque(int k)`: Initializes the deque with a maximum size of `k`.
- `boolean insertFront()`: Adds an item at the front of Deque. Returns true if the operation is successful, or false otherwise.
- `boolean insertLast()`: Adds an item at the rear of Deque. Returns true if the operation is successful, or false otherwise.
- `boolean deleteFront()`: Deletes an item from the front of Deque. Returns true if the operation is successful, or false otherwise.
- `boolean deleteLast()`: Deletes an item from the rear of Deque. Returns true if the operation is successful, or false otherwise.
- `int getFront()`: Returns the front item from the Deque. Returns -1 if the deque is empty.
- `int getRear()`: Returns the last item from the Deque. Returns -1 if the deque is empty.
- `boolean isEmpty()`: Returns true if the deque is empty, or false otherwise.
- `boolean isFull()`: Returns true if the deque is full, or false otherwise.

Example:

```
MyCircularDeque myCircularDeque = new
MyCircularDeque(3);
myCircularDeque.insertLast(1);  // return True
myCircularDeque.insertLast(2);  // return True
myCircularDeque.insertFront(3); // return True
myCircularDeque.insertFront(4); // return False (queue is
full)
myCircularDeque.getRear();      // return 2
myCircularDeque.isFull();       // return True
myCircularDeque.deleteLast();   // return True
myCircularDeque.insertFront(4); // return True
myCircularDeque.getFront();     // return 4
```

Solution 1: Array Implementation with Two Pointers

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularDeque {
private:
    vector<int> data;
    int front;
    int rear;
    int size;
    int capacity;

public:
    MyCircularDeque(int k) {
        data.resize(k);
        front = 0;
        rear = 0;
        size = 0;
        capacity = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;

        front = (front - 1 + capacity) % capacity;
        data[front] = value;
        size++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;

        data[rear] = value;
        rear = (rear + 1) % capacity;
        size++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;

        front = (front + 1) % capacity;
        size--;
        return true;
```

```
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        rear = (rear - 1 + capacity) % capacity;
        size--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return data[front];
    }

    int getRear() {
        if (isEmpty()) return -1;
        return data[(rear - 1 + capacity) % capacity];
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
    }
};
```

Solution 2: Array with Count and Head

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularDeque {
private:
    vector<int> data;
    int head;
    int count;
    int capacity;

public:
    MyCircularDeque(int k) {
        data.resize(k);
        head = 0;
        count = 0;
        capacity = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;

        head = (head - 1 + capacity) % capacity;
        data[head] = value;
        count++;
        return true;
    }
    bool insertLast(int value) {
        if (isFull()) return false;

        int tail = (head + count) % capacity;
        data[tail] = value;
        count++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;

        head = (head + 1) % capacity;
        count--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        count--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return data[head];
    }

    int getRear() {
        if (isEmpty()) return -1;
        int tail = (head + count - 1) % capacity;
        return data[tail];
    }

    bool isEmpty() {
        return count == 0;
    }

    bool isFull() {
        return count == capacity;
    }
};
```

Solution 3: Doubly Linked List Implementation

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class Node {
public:
    int val;
    Node* prev;
    Node* next;
    Node(int value) : val(value), prev(nullptr), next(nullptr)
{}
};

class MyCircularDeque {
private:
    Node* head;
    Node* tail;
    int size;
    int capacity;

public:
    MyCircularDeque(int k) {
        head = nullptr;
        tail = nullptr;
        size = 0;
        capacity = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;

        Node* newNode = new Node(value);
        if (isEmpty()) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        size++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;

        Node* newNode = new Node(value);
        if (isEmpty()) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        size++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;

        Node* temp = head;
        if (head == tail) { // Only one element
            head = tail = nullptr;
        } else {
            head = head->next;
            head->prev = nullptr;
        }
        delete temp;
        size--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        Node* temp = tail;
        if (head == tail) { // Only one element
            head = tail = nullptr;
        } else {
            tail = tail->prev;
            tail->next = nullptr;
        }
        delete temp;
        size--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return head->val;
    }

    int getRear() {
        if (isEmpty()) return -1;
        return tail->val;
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
    }

    ~MyCircularDeque() {
        while (!isEmpty()) {
            deleteFront();
        }
    }
};
```

Solution 4: Circular Doubly Linked List

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class Node {
public:
    int val;
    Node* prev;
    Node* next;
    Node(int value) : val(value), prev(nullptr), next(nullptr)
{}
};
```

```
class MyCircularDeque {
private:
    Node* head;
    Node* tail;
    int size;
    int capacity;

public:
    MyCircularDeque(int k) {
        head = tail = nullptr;
        size = 0;
        capacity = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;

        Node* newNode = new Node(value);
        if (isEmpty()) {
            head = tail = newNode;
            head->next = head;
            head->prev = head;
        } else {
            newNode->next = head;
            newNode->prev = tail;
            head->prev = newNode;
            tail->next = newNode;
            head = newNode;
        }
        size++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;

        Node* newNode = new Node(value);
        if (isEmpty()) {
            head = tail = newNode;
            head->next = head;
            head->prev = head;
        } else {
            newNode->prev = tail;
            newNode->next = head;
            tail->next = newNode;
            head->prev = newNode;
            tail = newNode;
        }
        size++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;

        if (size == 1) {
            delete head;
            head = tail = nullptr;
        } else {
            Node* temp = head;
            head = head->next;
            head->prev = tail;
            tail->next = head;
            delete temp;
        }
        size--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        if (size == 1) {
            delete tail;
            head = tail = nullptr;
        } else {
            Node* temp = tail;
            tail = tail->prev;
            tail->next = head;
            head->prev = tail;
            delete temp;
        }
        size--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return head->val;
    }

    int getRear() {
        if (isEmpty()) return -1;
        return tail->val;
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
    }

    ~MyCircularDeque() {
        while (!isEmpty()) {
            deleteFront();
        }
    }
};
```

Solution 5: Array with Wasted Space

Time Complexity: O(1) for all operations
Space Complexity: O(k+1)

```
class MyCircularDeque {
private:
    vector<int> data;
    int front;
    int rear;
    int capacity;

public:
    MyCircularDeque(int k) {
        data.resize(k + 1); // One extra space to distinguish
full vs empty
        front = 0;
        rear = 0;
        capacity = k + 1;
    }

    bool insertFront(int value) {
        if (isFull()) return false;

        front = (front - 1 + capacity) % capacity;
        data[front] = value;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;

        data[rear] = value;
        rear = (rear + 1) % capacity;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;

        front = (front + 1) % capacity;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        rear = (rear - 1 + capacity) % capacity;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return data[front];
    }

    int getRear() {
        if (isEmpty()) return -1;
        return data[(rear - 1 + capacity) % capacity];
    }

    bool isEmpty() {
        return front == rear;
    }

    bool isFull() {
        return (rear + 1) % capacity == front;
    }
};
```

Solution 6: Using Deque STL with Size Limit

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularDeque {
private:
    deque<int> dq;
    int maxSize;

public:
    MyCircularDeque(int k) {
        maxSize = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;
        dq.push_front(value);
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;
        dq.push_back(value);
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;
        dq.pop_front();
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;
        dq.pop_back();
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return dq.front();
    }
```

```
    int getRear() {
        if (isEmpty()) return -1;
        return dq.back();
    }

    bool isEmpty() {
        return dq.empty();
    }

    bool isFull() {
        return dq.size() == maxSize;
    }
};
```

Solution 7: Array with Size Tracking

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularDeque {
private:
    vector<int> data;
    int front;
    int rear;
    int currentSize;
    int maxSize;

public:
    MyCircularDeque(int k) {
        data.resize(k);
        front = 0;
        rear = k - 1; // Start rear at the end for symmetry
        currentSize = 0;
        maxSize = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;

        front = (front - 1 + maxSize) % maxSize;
        data[front] = value;
        currentSize++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;

        rear = (rear + 1) % maxSize;
        data[rear] = value;
        currentSize++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;
        front = (front + 1) % maxSize;
        currentSize--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        rear = (rear - 1 + maxSize) % maxSize;
        currentSize--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return data[front];
    }

    int getRear() {
        if (isEmpty()) return -1;
        return data[rear];
    }

    bool isEmpty() {
        return currentSize == 0;
    }

    bool isFull() {
        return currentSize == maxSize;
    }
};
```

Solution 8: Modular Arithmetic with Two Pointers

Time Complexity: O(1) for all operations
Space Complexity: O(k)

```
class MyCircularDeque {
private:
    vector<int> buffer;
    int front;
    int rear;
    int len;
    int cap;

public:
    MyCircularDeque(int k) {
        buffer.resize(k);
        front = 0;
        rear = 0;
        len = 0;
        cap = k;
    }

    bool insertFront(int value) {
        if (isFull()) return false;
        if (!isEmpty()) {
            front = (front - 1 + cap) % cap;
        }
        buffer[front] = value;
        len++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull()) return false;

        if (!isEmpty()) {
            rear = (rear + 1) % cap;
        }
        buffer[rear] = value;
        len++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty()) return false;

        if (len > 1) {
            front = (front + 1) % cap;
        }
        len--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty()) return false;

        if (len > 1) {
            rear = (rear - 1 + cap) % cap;
        }
        len--;
        return true;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return buffer[front];
    }

    int getRear() {
        if (isEmpty()) return -1;
        return buffer[rear];
    }

    bool isEmpty() {
        return len == 0;
    }

    bool isFull() {
        return len == cap;
    }
};
```

707. Problem: **Design Linked List**

Design your implementation of the linked list. You can choose to use a singly or doubly linked list.

A node in a singly linked list should have two attributes: `val` and `next`. `val` is the value of the current node, and `next` is a pointer/reference to the next node.

If you want to use the doubly linked list, you will need one more attribute `prev` to indicate the previous node in the linked list. Assume all nodes in the linked list are 0-indexed.

Implement the `MyLinkedList` class:
- `MyLinkedList()`: Initializes the `MyLinkedList` object.
- `int get(int index)`: Get the value of the `index-th` node in the linked list. If the index is invalid, return `-1`.
- `void addAtHead(int val)`: Add a node of value `val` before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
- `void addAtTail(int val)`: Append a node of value `val` as the last element of the linked list.
- `void addAtIndex(int index, int val)`: Add a node of value `val` before the `index-th` node in the linked list. If `index` equals the length of the linked list, the node will be appended to the end of the linked list. If `index` is greater than the length, the node will not be inserted.
- `void deleteAtIndex(int index)`: Delete the `index-th` node in the linked list, if the index is valid.

Example:

```
MyLinkedList myLinkedList = new MyLinkedList();
myLinkedList.addAtHead(1);
myLinkedList.addAtTail(3);
myLinkedList.addAtIndex(1, 2);    // linked list becomes 1->2->3
myLinkedList.get(1);              // return 2
myLinkedList.deleteAtIndex(1);    // now the linked list is 1->3
myLinkedList.get(1);              // return 3
```

Solution 1: Singly Linked List with Dummy Head

Time Complexity:
- get: O(n)
- addAtHead: O(1)
- addAtTail: O(n)
- addAtIndex: O(n)
- deleteAtIndex: O(n)

Space Complexity: O(n)

```
class MyLinkedList {
private:
    struct ListNode {
        int val;
        ListNode* next;
        ListNode(int x) : val(x), next(nullptr) {}
    };

    ListNode* dummy;
    int size;

public:
    MyLinkedList() {
        dummy = new ListNode(0);
        size = 0;
    }

    int get(int index) {
        if (index < 0 || index >= size) return -1;

        ListNode* curr = dummy->next;
        for (int i = 0; i < index; i++) {
            curr = curr->next;
        }
        return curr->val;
    }

    void addAtHead(int val) {
        addAtIndex(0, val);
    }

    void addAtTail(int val) {
        addAtIndex(size, val);
    }

    void addAtIndex(int index, int val) {
        if (index < 0 || index > size) return;

        ListNode* prev = dummy;
        for (int i = 0; i < index; i++) {
            prev = prev->next;
        }

        ListNode* newNode = new ListNode(val);
        newNode->next = prev->next;
        prev->next = newNode;
        size++;
    }

    void deleteAtIndex(int index) {
        if (index < 0 || index >= size) return;

        ListNode* prev = dummy;
        for (int i = 0; i < index; i++) {
            prev = prev->next;
        }
        ListNode* toDelete = prev->next;
        prev->next = toDelete->next;
        delete toDelete;
        size--;
    }

    ~MyLinkedList() {
        ListNode* curr = dummy;
        while (curr) {
            ListNode* temp = curr;
            curr = curr->next;
            delete temp;
        }
    }
};
```

---

solution 2: Doubly Linked List with Dummy Head and Tail

Time Complexity:
- get: O(n)
- addAtHead: O(1)
- addAtTail: O(1)
- addAtIndex: O(n)
- deleteAtIndex: O(n)

Space Complexity: O(n)

```
class MyLinkedList {
private:
    struct ListNode {
        int val;
        ListNode* prev;
        ListNode* next;
        ListNode(int x) : val(x), prev(nullptr), next(nullptr) {}
    };

    ListNode* head;
    ListNode* tail;
    int size;

public:
    MyLinkedList() {
        head = new ListNode(0); // dummy head
        tail = new ListNode(0); // dummy tail
        head->next = tail;
        tail->prev = head;
        size = 0;
    }

    int get(int index) {
        if (index < 0 || index >= size) return -1;

        ListNode* curr;
        if (index < size / 2) { // Optimize: start from head for
first half
            curr = head->next;
            for (int i = 0; i < index; i++) {
                curr = curr->next;
            }
        } else { // Start from tail for second half
            curr = tail->prev;
            for (int i = size - 1; i > index; i--) {
                curr = curr->prev;
            }
        }
        return curr->val;
    }

    void addAtHead(int val) {
        ListNode* newNode = new ListNode(val);
        newNode->next = head->next;
        newNode->prev = head;
        head->next->prev = newNode;
        head->next = newNode;
        size++;
    }

    void addAtTail(int val) {
        ListNode* newNode = new ListNode(val);
        newNode->prev = tail->prev;
        newNode->next = tail;
        tail->prev->next = newNode;
        tail->prev = newNode;
        size++;
    }

    void addAtIndex(int index, int val) {
        if (index < 0 || index > size) return;

        ListNode* prev, *next;
        if (index < size / 2) {
            prev = head;
            for (int i = 0; i < index; i++) {
                prev = prev->next;
            }
            next = prev->next;
        } else {
            next = tail;
            for (int i = size; i > index; i--) {
                next = next->prev;
            }
            prev = next->prev;
        }

        ListNode* newNode = new ListNode(val);
        newNode->prev = prev;
        newNode->next = next;
        prev->next = newNode;
        next->prev = newNode;
        size++;
    }

    void deleteAtIndex(int index) {
        if (index < 0 || index >= size) return;

        ListNode* toDelete;
        if (index < size / 2) {
            toDelete = head->next;
            for (int i = 0; i < index; i++) {
                toDelete = toDelete->next;
            }
        } else {
            toDelete = tail->prev;
            for (int i = size - 1; i > index; i--) {
                toDelete = toDelete->prev;
            }
        }

        toDelete->prev->next = toDelete->next;
        toDelete->next->prev = toDelete->prev;
        delete toDelete;
        size--;
    }

    ~MyLinkedList() {
        ListNode* curr = head;
        while (curr) {
            ListNode* temp = curr;
            curr = curr->next;
            delete temp;
        }
    }
};
```

Solution 3: Singly Linked List with Tail Pointer

Time Complexity:
- get: O(n)
- addAtHead: O(1)
- addAtTail: O(1)
- addAtIndex: O(n)
- deleteAtIndex: O(n)

Space Complexity: O(n)

```
class MyLinkedList {
private:
    struct ListNode {
        int val;
        ListNode* next;
        ListNode(int x) : val(x), next(nullptr) {}
    };

    ListNode* head;
    ListNode* tail;
    int size;

public:
    MyLinkedList() {
        head = nullptr;
        tail = nullptr;
```

```
    size = 0;
}

int get(int index) {
    if (index < 0 || index >= size) return -1;

    ListNode* curr = head;
    for (int i = 0; i < index; i++) {
        curr = curr->next;
    }
    return curr->val;
}

void addAtHead(int val) {
    ListNode* newNode = new ListNode(val);
    newNode->next = head;
    head = newNode;
    if (size == 0) {
        tail = head;
    }
    size++;
}

void addAtTail(int val) {
    ListNode* newNode = new ListNode(val);
    if (size == 0) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        tail = newNode;
    }
    size++;
}

void addAtIndex(int index, int val) {
    if (index < 0 || index > size) return;

    if (index == 0) {
        addAtHead(val);
    } else if (index == size) {
        addAtTail(val);
    } else {
        ListNode* prev = head;
        for (int i = 0; i < index - 1; i++) {
            prev = prev->next;
        }
        ListNode* newNode = new ListNode(val);
        newNode->next = prev->next;
        prev->next = newNode;
        size++;
    }
}

void deleteAtIndex(int index) {
    if (index < 0 || index >= size) return;

    if (index == 0) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
        if (size == 1) {
            tail = nullptr;
        }
    } else {
        ListNode* prev = head;
        for (int i = 0; i < index - 1; i++) {
            prev = prev->next;
        }
        ListNode* toDelete = prev->next;
        prev->next = toDelete->next;
        if (index == size - 1) {
            tail = prev;
        }
        delete toDelete;
    }
    size--;
}

~MyLinkedList() {
    ListNode* curr = head;
    while (curr) {
        ListNode* temp = curr;
        curr = curr->next;
        delete temp;
    }
}
};
```

Solution 4: Vector-Based Implementation

Time Complexity:
- get: O(1)
- addAtHead: O(n)
- addAtTail: O(1) amortized
- addAtIndex: O(n)
- deleteAtIndex: O(n)
Space Complexity: O(n)

```
class MyLinkedList {
private:
    vector<int> data;

public:
    MyLinkedList() {}

    int get(int index) {
        if (index < 0 || index >= data.size()) return -1;
        return data[index];
    }

    void addAtHead(int val) {
        data.insert(data.begin(), val);
    }

    void addAtTail(int val) {
        data.push_back(val);
    }

    void addAtIndex(int index, int val) {
        if (index < 0 || index > data.size()) return;
        data.insert(data.begin() + index, val);
    }

    void deleteAtIndex(int index) {
        if (index < 0 || index >= data.size()) return;
        data.erase(data.begin() + index);
    }
};
```

Solution 5: Circular Singly Linked List

Time Complexity:
- get: O(n)
- addAtHead: O(1)
- addAtTail: O(1)
- addAtIndex: O(n)
- deleteAtIndex: O(n)
Space Complexity: O(n)

```
class MyLinkedList {
private:
    struct ListNode {
        int val;
        ListNode* next;
        ListNode(int x) : val(x), next(nullptr) {}
    };

    ListNode* tail; // In circular list, tail->next is head
    int size;

public:
    MyLinkedList() {
        tail = nullptr;
        size = 0;
    }

    int get(int index) {
        if (index < 0 || index >= size) return -1;

        ListNode* curr = tail->next; // head
        for (int i = 0; i < index; i++) {
            curr = curr->next;
        }
        return curr->val;
    }

    void addAtHead(int val) {
        ListNode* newNode = new ListNode(val);
        if (size == 0) {
            tail = newNode;
            tail->next = tail; // circular reference
        } else {
            newNode->next = tail->next;
            tail->next = newNode;
        }
        size++;
    }

    void addAtTail(int val) {
        ListNode* newNode = new ListNode(val);
        if (size == 0) {
            tail = newNode;
            tail->next = tail;
        } else {
            newNode->next = tail->next;
            tail->next = newNode;
            tail = newNode;
        }
        size++;
    }

    void addAtIndex(int index, int val) {
        if (index < 0 || index > size) return;

        if (index == 0) {
            addAtHead(val);
        } else if (index == size) {
            addAtTail(val);
        } else {
            ListNode* prev = tail->next; // head
            for (int i = 0; i < index - 1; i++) {
                prev = prev->next;
            }
            ListNode* newNode = new ListNode(val);
            newNode->next = prev->next;
            prev->next = newNode;
            size++;
        }
    }

    void deleteAtIndex(int index) {
        if (index < 0 || index >= size) return;

        if (size == 1) {
            delete tail;
            tail = nullptr;
        } else {
            ListNode* prev = tail->next; // head
            for (int i = 0; i < index - 1; i++) {
                prev = prev->next;
            }
            ListNode* toDelete = prev->next;
            prev->next = toDelete->next;
            if (toDelete == tail) {
                tail = prev;
            }
            delete toDelete;
        }
```

```
        size--;
    }

    ~MyLinkedList() {
        if (size == 0) return;

        ListNode* curr = tail->next;
        while (curr != tail) {
            ListNode* temp = curr;
            curr = curr->next;
            delete temp;
        }
        delete tail;
    }
};
```

Solution 6: Optimized Doubly Linked List with Size

Time Complexity:
- get: O(min(index, size-index))
- addAtHead: O(1)
- addAtTail: O(1)
- addAtIndex: O(min(index, size-index))
- deleteAtIndex: O(min(index, size-index))
Space Complexity: O(n)

```
class MyLinkedList {
private:
    struct ListNode {
        int val;
        ListNode* prev;
        ListNode* next;
        ListNode(int x) : val(x), prev(nullptr), next(nullptr) {}
    };

    ListNode* head;
    ListNode* tail;
    int size;

public:
    MyLinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    int get(int index) {
        if (index < 0 || index >= size) return -1;

        ListNode* curr;
        if (index < size / 2) {
            curr = head;
            for (int i = 0; i < index; i++) {
                curr = curr->next;
            }
        } else {
            curr = tail;
            for (int i = size - 1; i > index; i--) {
                curr = curr->prev;
            }
        }
        return curr->val;
    }

    void addAtHead(int val) {
        ListNode* newNode = new ListNode(val);
        if (size == 0) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        size++;
    }

    void addAtTail(int val) {
        ListNode* newNode = new ListNode(val);
        if (size == 0) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        size++;
    }

    void addAtIndex(int index, int val) {
        if (index < 0 || index > size) return;

        if (index == 0) {
            addAtHead(val);
        } else if (index == size) {
            addAtTail(val);
        } else {
            ListNode* curr;
            if (index < size / 2) {
                curr = head;
                for (int i = 0; i < index; i++) {
                    curr = curr->next;
                }
            } else {
                curr = tail;
                for (int i = size - 1; i > index; i--) {
                    curr = curr->prev;
                }
            }

            ListNode* newNode = new ListNode(val);
            newNode->prev = curr->prev;
            newNode->next = curr;
            curr->prev->next = newNode;
            curr->prev = newNode;
            size++;
        }
    }

    void deleteAtIndex(int index) {
        if (index < 0 || index >= size) return;

        if (size == 1) {
            delete head;
            head = tail = nullptr;
        } else if (index == 0) {
            ListNode* temp = head;
            head = head->next;
            head->prev = nullptr;
            delete temp;
        } else if (index == size - 1) {
            ListNode* temp = tail;
            tail = tail->prev;
            tail->next = nullptr;
            delete temp;
        } else {
            ListNode* curr;
            if (index < size / 2) {
                curr = head;
                for (int i = 0; i < index; i++) {
                    curr = curr->next;
                }
            } else {
                curr = tail;
                for (int i = size - 1; i > index; i--) {
                    curr = curr->prev;
                }
            }

            curr->prev->next = curr->next;
            curr->next->prev = curr->prev;
            delete curr;
        }
        size--;
    }

    ~MyLinkedList() {
        ListNode* curr = head;
        while (curr) {
            ListNode* temp = curr;
            curr = curr->next;
            delete temp;
        }
    }
};
```

817. Problem: Linked List Components

You are given the head of a linked list containing unique integer values and an integer array `nums` which is a subset of linked list values. Return the number of connected components in `nums` where two values are connected if they appear consecutively in the linked list.

Example 1:
Input: `head = [0,1,2,3], nums = [0,1,3]`
Output: `2`
Explanation: 0 and 1 are connected, so [0,1] and [3] are two connected components.

Example 2:
Input: `head = [0,1,2,3,4], nums = [0,3,1,4]`
Output: `2`
Explanation: 0 and 1 are connected, and 3 and 4 are connected, so [0,1] and [3,4] are two connected components.

Solution 1: Set with Component Detection

Time Complexity: O(n)
Space Complexity: O(m) where m is the size of nums

```
class Solution {
public:
    int numComponents(ListNode* head, vector<int>& nums) {
        unordered_set<int> numSet(nums.begin(), nums.end());
        int components = 0;
        bool inComponent = false;

        ListNode* curr = head;
        while (curr) {
            if (numSet.count(curr->val)) {
                if (!inComponent) {
                    components++;
                    inComponent = true;
                }
            } else {
                inComponent = false;
            }
            curr = curr->next;
        }

        return components;
    }
};
```

Solution 2: Set with Previous Tracking

Time Complexity: O(n)
Space Complexity: O(m)

```
class Solution {
public:
    int numComponents(ListNode* head, vector<int>&
nums) {
        unordered_set<int> numSet(nums.begin(),
nums.end());
        int components = 0;

        ListNode* curr = head;
        while (curr) {
            // Start of a new component when current is in
nums and
            // (it's head OR previous is not in nums)
            if (numSet.count(curr->val) &&
                (!curr->next || !numSet.count(curr->next->val)))
{
                components++;
            }
            curr = curr->next;
        }

        return components;
    }
};
```

Solution 3: Using Array for Faster Lookup

Time Complexity: O(n)
Space Complexity: O(10001) = O(1) since values are 0 <= Node.val <= 10000

```
class Solution {
public:
    int numComponents(ListNode* head, vector<int>&
nums) {
        vector<bool> present(10001, false);
        for (int num : nums) {
            present[num] = true;
        }
        int components = 0;
        ListNode* curr = head;

        while (curr) {
            if (present[curr->val]) {
                components++;
                // Skip all consecutive nodes in the same
component
                while (curr && curr->next &&
present[curr->next->val]) {
                    curr = curr->next;
                }
            }
            curr = curr->next;
        }
        return components;
    }
};
```

Solution 4: Group Detection Approach

Time Complexity: O(n)
Space Complexity: O(m)

```
class Solution {
public:
    int numComponents(ListNode* head, vector<int>&
nums) {
        unordered_set<int> numSet(nums.begin(),
nums.end());
        int components = 0;
        bool counting = false;

        ListNode* curr = head;
        while (curr) {
            if (numSet.count(curr->val)) {
                if (!counting) {
                    components++;
                    counting = true;
                }
            } else {
                counting = false;
            }
            curr = curr->next;
        }

        return components;
    }
};
```

Solution 5: Two Pointers with Set

Time Complexity: O(n)
Space Complexity: O(m)

```
class Solution {
public:
    int numComponents(ListNode* head, vector<int>&
nums) {
        unordered_set<int> numSet(nums.begin(),
nums.end());
        int components = 0;

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast) {
            // Move fast until we find a node not in nums
            while (fast && numSet.count(fast->val)) {
                fast = fast->next;
            }

            // If slow moved, we found a component
            if (slow != fast) {
                components++;
            }
            // Move both pointers to next position
            if (fast) {
                fast = fast->next;
            }
            slow = fast;
        }

        return components;
    }
};
```

876. Problem: **Middle of the Linked List**

Given the head of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

Example 1:
Input: `head = [1,2,3,4,5]`
Output: `[3,4,5]`
Explanation: The middle node of the list is node 3.

Example 2:
Input: `head = [1,2,3,4,5,6]`
Output: `[4,5,6]`
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.

Solution 1: Slow and Fast Pointers (Tortoise and Hare)

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

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        return slow;
    }
};
```

Solution 2: Two Pass - Count then Find

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

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        int length = 0;
        ListNode* curr = head;

        // First pass: count nodes
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Second pass: find middle
        curr = head;
        for (int i = 0; i < length / 2; i++) {
            curr = curr->next;
        }

        return curr;
    }
};
```

Solution 3: Using Vector/Array

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

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        vector<ListNode*> nodes;
        ListNode* curr = head;

        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        return nodes[nodes.size() / 2];
    }
};
```

Solution 4: Recursive with Counter

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        int length = getLength(head);
        return getNodeAt(head, length / 2);
    }
private:
    int getLength(ListNode* head) {
        if (!head) return 0;
        return 1 + getLength(head->next);
```

```
    }

    ListNode* getNodeAt(ListNode* head, int index) {
        if (index == 0) return head;
        return getNodeAt(head->next, index - 1);
    }
};
```

Solution 5: One Pass with Two Pointers (Alternative)

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

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* mid = head;
        ListNode* curr = head;
        int count = 0;

        while (curr) {
            // Move mid every 2 steps
            if (count % 2 == 1) {
                mid = mid->next;
            }
            count++;
            curr = curr->next;
        }

        return mid;
    }
};
```

1019. Problem: **Next Greater Node In Linked List**

You are given the head of a linked list with n nodes. For each node in the list, find the value of the next greater node. That is, for each node, find the value of the first node that is next to it and has a strictly larger value than it.

Return an integer array answer where `answer[i]` is the value of the next greater node of the `i-th` node. If there is no next greater node, return `0` for that node.

Example 1:
Input: `head = [2,1,5]`
Output: `[5,5,0]`
Explanation:
- Node 0: next greater value is 5
- Node 1: next greater value is 5
- Node 2: no next greater value

Example 2:
Input: `head = [2,7,4,3,5]`
Output: `[7,0,5,5,0]`

Solution 1: Monotonic Stack with Vector

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

```
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        vector<int> values;
        ListNode* curr = head;

        // Convert linked list to array
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        vector<int> result(values.size(), 0);
        stack<int> st; // stack stores indices

        for (int i = 0; i < values.size(); i++) {
            // While stack is not empty and current value >
value at stack top
            while (!st.empty() && values[i] > values[st.top()]) {
                result[st.top()] = values[i];
                st.pop();
            }
            st.push(i);
        }

        return result;
    }
};
```

Solution 2: Monotonic Stack with Pair (Value, Index)

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

```
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        vector<int> result;
        stack<pair<int, int>> st; // pair: {value, index}

        ListNode* curr = head;
        int index = 0;

        while (curr) {
            result.push_back(0); // Initialize with 0

            // Process stack: current value is next greater for
previous smaller elements
            while (!st.empty() && curr->val > st.top().first) {
                result[st.top().second] = curr->val;
                st.pop();
            }
            st.push({curr->val, index});
            index++;
            curr = curr->next;
        }

        return result;
    }
};
```

Solution 3: Reverse and Use Stack

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

```
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        // First, reverse the linked list
        ListNode* prev = nullptr;
        ListNode* curr = head;
        int length = 0;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
            length++;
        }

        // Now traverse reversed list and use stack
        vector<int> result(length, 0);
        stack<int> st;
        curr = prev;
        int i = length - 1;

        while (curr) {
            // Pop elements smaller than current
            while (!st.empty() && st.top() <= curr->val) {
                st.pop();
            }

            // If stack has larger element, it's the next greater
            if (!st.empty()) {
                result[i] = st.top();
            }

            st.push(curr->val);
            curr = curr->next;
            i--;
        }

        return result;
    }
};
```

Solution 4: Brute Force (Naive)

Time Complexity: O(n²)
Space Complexity: O(n)

```
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        vector<int> values;
        ListNode* curr = head;

        // Convert to array
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        vector<int> result(values.size(), 0);

        for (int i = 0; i < values.size(); i++) {
            for (int j = i + 1; j < values.size(); j++) {
                if (values[j] > values[i]) {
                    result[i] = values[j];
                    break;
                }
            }
        }

        return result;
    }
};
```

Solution 5: Recursive with Global Variables

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
private:
    vector<int> result;
    stack<int> st;
    int index = 0;

public:
    vector<int> nextLargerNodes(ListNode* head) {
        if (!head) return {};

        // First pass to get size and initialize result
        ListNode* curr = head;
        int size = 0;
        while (curr) {
            size++;
            curr = curr->next;
        }
        result.resize(size, 0);

        // Recursive processing
```

```
        process(head);
        return result;
    }

private:
    void process(ListNode* node) {
        if (!node) return;

        // Process recursively first (go to end)
        process(node->next);

        // Process current node
        while (!st.empty() && st.top() <= node->val) {
            st.pop();
        }

        if (!st.empty()) {
            result[index] = st.top();
        }

        st.push(node->val);
        index++;
    }
};
```

1171 Problem: **Remove Zero Sum Consecutive Nodes from Linked List**

Given the head of a linked list, we repeatedly delete consecutive sequences of nodes that sum to 0 until there are no such sequences. After doing so, return the head of the final linked list.

Example 1:
Input: `head = [1,2,-3,3,1]`
Output: `[3,1]`
Explanation: `[1,2,-3]` sums to 0, so we remove it.

Example 2:
Input: `head = [1,2,3,-3,4]`
Output: `[1,2,4]`

Example 3:
Input: `head = [1,2,3,-3,-2]`
Output: `[1]`

Solution 1: Prefix Sum with HashMap

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

```
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        unordered_map<int, ListNode*> prefixSum;
        int sum = 0;
        prefixSum[0] = dummy;

        ListNode* curr = head;
        while (curr) {
            sum += curr->val;

            // If we've seen this sum before, remove the nodes between
            if (prefixSum.find(sum) != prefixSum.end()) {
                ListNode* prev = prefixSum[sum];
                ListNode* temp = prev->next;
                int tempSum = sum;

                // Remove nodes from the map that are between prev and curr
                while (temp != curr) {
                    tempSum += temp->val;
                    prefixSum.erase(tempSum);
                    temp = temp->next;
                }

                prev->next = curr->next;
            } else {
                prefixSum[sum] = curr;
            }

            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 2: Two Pass with Prefix Sum

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

```
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        unordered_map<int, ListNode*> prefixSum;
        prefixSum[0] = dummy;

        int sum = 0;
        ListNode* curr = head;

        // First pass: build prefix sum map
        while (curr) {
            sum += curr->val;
            prefixSum[sum] = curr;
            curr = curr->next;
        }

        // Second pass: remove zero sum sequences
        sum = 0;
        curr = dummy;
        while (curr) {
            sum += curr->val;
            curr->next = prefixSum[sum]->next;
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 3: Brute Force - Check All Subarrays

Time Complexity: O(n²)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        bool found = true;

        // Keep removing until no zero sum sequences found
        while (found) {
            found = false;
            unordered_map<int, ListNode*> prefixSum;
            prefixSum[0] = dummy;

            int sum = 0;
            ListNode* curr = dummy->next;

            while (curr) {
                sum += curr->val;

                if (prefixSum.find(sum) != prefixSum.end()) {
                    // Remove the sequence
                    ListNode* prev = prefixSum[sum];
                    prev->next = curr->next;
                    found = true;
                    break;
                } else {
                    prefixSum[sum] = curr;
                }
                curr = curr->next;
            }
        }

        return dummy->next;
    }
};
```

Solution 4: Recursive Approach

Time Complexity: O(n²)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        if (!head) return nullptr;

        // Try to find zero sum sequence starting from head
        int sum = 0;
        ListNode* curr = head;

        while (curr) {
            sum += curr->val;
            if (sum == 0) {
                // Entire sequence from head to curr sums to zero
                return removeZeroSumSublists(curr->next);
            }
            curr = curr->next;
        }

        // No zero sum starting from head, keep head and process rest
        head->next = removeZeroSumSublists(head->next);
        return head;
    }
};
```

Solution 5: Iterative with Multiple Passes

Time Complexity: O(n²)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        bool modified = true;

        while (modified) {
            modified = false;
            unordered_map<int, ListNode*> prefixSum;
            prefixSum[0] = dummy;

            int sum = 0;
            ListNode* curr = dummy->next;

            while (curr) {
                sum += curr->val;

                if (prefixSum.count(sum)) {
```

```
                // Remove sequence from prefixSum[sum]->next to curr
                ListNode* prev = prefixSum[sum];
                prev->next = curr->next;
                modified = true;
                break;
            }

            prefixSum[sum] = curr;
            curr = curr->next;
        }
    }

    return dummy->next;
  }
};
```

1290. Problem: **Convert Binary Number in a Linked List to Integer**

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number. Return the decimal value of the number in the linked list.

Example 1:
Input: `head = [1,0,1]`
Output: `5`
Explanation: `(101) in base 2 = (5) in base 10`

Example 2:
Input: `head = [0]`
Output: `0`

Example 3:
Input: `head = [1]`
Output: `1`

Example 4:
Input: `head = [1,0,0,1,0,0,1,1,1,0,0,0,0,0,0]`
Output: `18880`

Solution 1: Left Shift (Bit Manipulation)

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

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        int result = 0;
        ListNode* curr = head;

        while (curr) {
            result = (result << 1) | curr->val;
            curr = curr->next;
        }

        return result;
    }
};
```

Solution 2: Mathematical Approach

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

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        int result = 0;
        ListNode* curr = head;

        while (curr) {
            result = result * 2 + curr->val;
            curr = curr->next;
        }

        return result;
    }
};
```

Solution 3: Reverse and Calculate

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

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        // First, reverse the linked list
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        // Now calculate from LSB to MSB
        int result = 0;
        int power = 1;
        curr = prev;

        while (curr) {
            result += curr->val * power;
            power *= 2;
            curr = curr->next;
        }

        return result;
```

Solution 4: Using Vector/Array

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

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        vector<int> bits;
        ListNode* curr = head;

        // Store all bits in vector
        while (curr) {
            bits.push_back(curr->val);
            curr = curr->next;
        }

        // Calculate decimal value
        int result = 0;
        int n = bits.size();

        for (int i = 0; i < n; i++) {
            if (bits[i] == 1) {
                result += (1 << (n - i - 1));
            }
        }
        return result;
    }
};
```

Solution 5: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        return helper(head, 0);
    }

private:
    int helper(ListNode* node, int current) {
        if (!node) return current;
        return helper(node->next, (current << 1) | node->val);
    }
};
```

Solution 6: Two Pass - Count then Calculate

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

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        // First pass: count length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Second pass: calculate decimal value
        int result = 0;
        curr = head;
        int power = length - 1;

        while (curr) {
            if (curr->val == 1) {
                result += (1 << power);
            }
            power--;
            curr = curr->next;
        }
        return result;
    }
};
```

Solution 7: Using String Conversion

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

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        string binary = "";
        ListNode* curr = head;

        while (curr) {
            binary += to_string(curr->val);
            curr = curr->next;
        }

        // Convert binary string to decimal
        return stoi(binary, nullptr, 2);
    }
};
```

1669. Problem: **Merge In Between Linked Lists**

You are given two linked lists: `list1` and `list2` of sizes `n` and `m` respectively. Remove the `a-th` to `b-th` nodes from `list1` and put `list2` in their place.

Example 1:
Input: `list1 = [0,1,2,3,4,5], a = 3, b = 4, list2 = [1000000,1000001,1000002]`
Output: `[0,1,2,1000000,1000001,1000002,5]`

Explanation: We remove nodes 3 and 4 and put list2 in their place.

Example 2:
Input: `list1 = [0,1,2,3,4,5,6], a = 2, b = 5, list2 = [1000000,1000001,1000002,1000003,1000004]`
Output:
`[0,1,1000000,1000001,1000002,1000003,1000004,6]`

Solution 1: Find Cut Points and Connect

Time Complexity: O(n + m)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
        ListNode* dummy = new ListNode(0);
        dummy->next = list1;

        // Find the node before a (at position a-1)
        ListNode* beforeA = dummy;
        for (int i = 0; i < a; i++) {
            beforeA = beforeA->next;
        }

        // Find the node after b (at position b+1)
        ListNode* afterB = beforeA;
        for (int i = a; i <= b + 1; i++) {
            afterB = afterB->next;
        }

        // Find the tail of list2
        ListNode* tail2 = list2;
        while (tail2->next) {
            tail2 = tail2->next;
        }

        // Connect beforeA to list2, and tail2 to afterB
        beforeA->next = list2;
        tail2->next = afterB;

        return dummy->next;
    }
};
```

Solution 2: Two Pointers without Dummy

Time Complexity: O(n + m)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
        // Find the node before position a
        ListNode* prevA = list1;
        for (int i = 0; i < a - 1; i++) {
            prevA = prevA->next;
        }

        // Find the node at position b
        ListNode* nodeB = prevA;
        for (int i = a - 1; i <= b; i++) {
            nodeB = nodeB->next;
        }

        // Find tail of list2
        ListNode* tail2 = list2;
        while (tail2->next) {
            tail2 = tail2->next;
        }

        // Connect prevA to list2 and tail2 to nodeB
        prevA->next = list2;
        tail2->next = nodeB;

        return list1;
    }
};
```

Solution 3: Using Separate Pointers

Time Complexity: O(n + m)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
        ListNode* start = list1;
        ListNode* end = list1;

        // Move start to node before a (position a-1)
        for (int i = 0; i < a - 1; i++) {
            start = start->next;
        }
        // Move end to node at b (position b)
        for (int i = 0; i < b; i++) {
            end = end->next;
        }
        // Find tail of list2
        ListNode* tail2 = list2;
        while (tail2->next) {
            tail2 = tail2->next;
        }
        // Connect start to list2 and tail2 to end->next
        start->next = list2;
        tail2->next = end->next;
        return list1;
    }
};
```

Solution 4: Count and Replace

Time Complexity: O(n + m)
Space Complexity: O(1)

```
class Solution {
public:
    ListNode* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
        ListNode* curr = list1;
        int count = 0;

        // Find node before a
        ListNode* beforeA = nullptr;
        while (count < a - 1) {
            curr = curr->next;
            count++;
        }
        beforeA = curr;

        // Find node after b
        while (count <= b) {
            curr = curr->next;
            count++;
        }
        ListNode* afterB = curr;

        // Find tail of list2
        ListNode* tail2 = list2;
        while (tail2->next) {
            tail2 = tail2->next;
        }

        // Connect the lists
        beforeA->next = list2;
        tail2->next = afterB;

        return list1;
    }
};
```

Solution 5: Recursive Approach (Alternative)

Time Complexity: O(n + m)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
        return helper(list1, a, b, list2, 0);
    }

private:
    ListNode* helper(ListNode* node, int a, int b, ListNode* list2, int index) {
        if (!node) return nullptr;
        if (index == a - 1) {
            // Found node before removal section
            ListNode* tail2 = list2;
            while (tail2->next) {
                tail2 = tail2->next;
            }

            // Skip to node after b
            ListNode* afterB = node;
            for (int i = a - 1; i <= b; i++) {
                afterB = afterB->next;
            }

            node->next = list2;
            tail2->next = afterB;
            return node;
        }

        node->next = helper(node->next, a, b, list2, index + 1);
        return node;
    }
};
```

1670. Problem: **Design Front Middle Back Queue**

Design a queue that supports push and pop operations in the front, middle, and back.

Implement the `FrontMiddleBack` class:
- `FrontMiddleBack()`: Initializes the queue.
- `void pushFront(int val)`: Adds val to the front of the queue.
- `void pushMiddle(int val)`: Adds val to the middle of the queue.
- `void pushBack(int val)`: Adds val to the back of the queue.
- `int popFront()`: Removes the front element and returns it. If the queue is empty, return -1.
- `int popMiddle()`: Removes the middle element and returns it. If the queue is empty, return -1.
- `int popBack()`: Removes the back element and returns it. If the queue is empty, return -1.

Note: The middle is the middle of the queue in terms of the number of elements. For even number of elements, the middle is the first middle element.

Example:

Input:
["FrontMiddleBackQueue", "pushFront", "pushBack", "pushMiddle", "pushMiddle", "popFront", "popMiddle", "popMiddle", "popBack", "popFront"]
[[], [1], [2], [3], [4], [], [], [], [], []]
Output:
[null, null, null, null, null, 1, 3, 4, 2, -1]

Solution 1: Two Deques (Optimal)

Time Complexity: O(1) for all operations
Space Complexity: O(n)

```
class FrontMiddleBackQueue {
private:
    deque<int> first, second;

    void balance() {
        // Ensure first.size() == second.size() or first.size()
== second.size() + 1
        if (first.size() > second.size() + 1) {
            second.push_front(first.back());
            first.pop_back();
        } else if (first.size() < second.size()) {
            first.push_back(second.front());
            second.pop_front();
        }
    }

public:
    FrontMiddleBackQueue() {}

    void pushFront(int val) {
        first.push_front(val);
        balance();
    }

    void pushMiddle(int val) {
        if (first.size() > second.size()) {
            second.push_front(first.back());
            first.pop_back();
        }
        first.push_back(val);
    }

    void pushBack(int val) {
        second.push_back(val);
        balance();
    }

    int popFront() {
        if (first.empty()) return -1;
        int val = first.front();
        first.pop_front();
        balance();
        return val;
    }

    int popMiddle() {
        if (first.empty()) return -1;
        int val = first.back();
        first.pop_back();
        balance();
        return val;
    }

    int popBack() {
        if (first.empty() && second.empty()) return -1;
        int val;
        if (second.empty()) {
            val = first.back();
            first.pop_back();
        } else {
            val = second.back();
            second.pop_back();
        }
        balance();
        return val;
    }
};
```

Solution 2: Single Deque

Time Complexity: O(n) for middle operations, O(1) for front/back
Space Complexity: O(n)

```
class FrontMiddleBackQueue {
private:
    deque<int> dq;

public:
    FrontMiddleBackQueue() {}

    void pushFront(int val) {
        dq.push_front(val);
    }

    void pushMiddle(int val) {
        int mid = dq.size() / 2;
        dq.insert(dq.begin() + mid, val);
    }

    void pushBack(int val) {
        dq.push_back(val);
    }

    int popFront() {
        if (dq.empty()) return -1;
        int val = dq.front();
        dq.pop_front();
        return val;
    }

    int popMiddle() {
        if (dq.empty()) return -1;
        int mid = (dq.size() - 1) / 2;
        int val = dq[mid];
        dq.erase(dq.begin() + mid);
        return val;
    }

    int popBack() {
        if (dq.empty()) return -1;
        int val = dq.back();
        dq.pop_back();
        return val;
    }
};
```

Solution 3: Two Stacks

Time Complexity: O(n) for middle operations
Space Complexity: O(n)

```
class FrontMiddleBackQueue {
private:
    stack<int> front, back;

    void balance() {
        // Ensure front.size() >= back.size() and difference
<= 1
        if (front.size() > back.size() + 1) {
            back.push(front.top());
            front.pop();
        } else if (front.size() < back.size()) {
            front.push(back.top());
            back.pop();
        }
    }

public:
    FrontMiddleBackQueue() {}

    void pushFront(int val) {
        front.push(val);
        balance();
    }

    void pushMiddle(int val) {
        if (front.size() > back.size()) {
            back.push(front.top());
            front.pop();
        }
        front.push(val);
    }

    void pushBack(int val) {
        // Reverse back to push to actual back
        stack<int> temp;
        while (!back.empty()) {
            temp.push(back.top());
            back.pop();
        }
        temp.push(val);
        while (!temp.empty()) {
            back.push(temp.top());
            temp.pop();
        }
        balance();
    }

    int popFront() {
        if (front.empty()) return -1;
        int val = front.top();
        front.pop();
        balance();
        return val;
    }

    int popMiddle() {
        if (front.empty()) return -1;
        int val = front.top();
        front.pop();
        balance();
        return val;
    }

    int popBack() {
        if (front.empty() && back.empty()) return -1;
        if (back.empty()) {
            int val = front.top();
            front.pop();
            balance();
            return val;
        }
        // Reverse back to pop from actual back
        stack<int> temp;
        while (!back.empty()) {
            temp.push(back.top());
            back.pop();
        }
        int val = temp.top();
        temp.pop();
        while (!temp.empty()) {
            back.push(temp.top());
            temp.pop();
        }
        balance();
        return val;
    }
};
```

Solution 4: Vector Implementation

Time Complexity: O(n) for middle operations
Space Complexity: O(n)

```
class FrontMiddleBackQueue {
private:
    vector<int> vec;

public:
    FrontMiddleBackQueue() {}

    void pushFront(int val) {
        vec.insert(vec.begin(), val);
```

```cpp
    }

    void pushMiddle(int val) {
        int mid = vec.size() / 2;
        vec.insert(vec.begin() + mid, val);
    }

    void pushBack(int val) {
        vec.push_back(val);
    }

    int popFront() {
        if (vec.empty()) return -1;
        int val = vec[0];
        vec.erase(vec.begin());
        return val;
    }

    int popMiddle() {
        if (vec.empty()) return -1;
        int mid = (vec.size() - 1) / 2;
        int val = vec[mid];
        vec.erase(vec.begin() + mid);
        return val;
    }

    int popBack() {
        if (vec.empty()) return -1;
        int val = vec.back();
        vec.pop_back();
        return val;
    }
};
```

Solution 5: Doubly Linked List

Time Complexity: O(n) for middle operations
Space Complexity: O(n)

```cpp
class Node {
public:
    int val;
    Node* prev;
    Node* next;
    Node(int v) : val(v), prev(nullptr), next(nullptr) {}
};

class FrontMiddleBackQueue {
private:
    Node* head;
    Node* tail;
    int size;

    Node* getMiddleNode() {
        Node* slow = head;
        Node* fast = head;
        while (fast && fast->next && fast->next->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }

public:
    FrontMiddleBackQueue() : head(nullptr), tail(nullptr),
size(0) {}

    void pushFront(int val) {
        Node* newNode = new Node(val);
        if (size == 0) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        size++;
    }

    void pushMiddle(int val) {
        if (size == 0) {
            pushFront(val);
            return;
        }

        Node* middle = getMiddleNode();
        Node* newNode = new Node(val);

        if (size % 2 == 0) {
            // Insert after middle
            newNode->next = middle->next;
            newNode->prev = middle;
            if (middle->next) middle->next->prev = newNode;
            middle->next = newNode;
            if (middle == tail) tail = newNode;
        } else {
            // Insert before middle
            newNode->prev = middle->prev;
            newNode->next = middle;
            if (middle->prev) middle->prev->next = newNode;
            middle->prev = newNode;
            if (middle == head) head = newNode;
        }
        size++;
    }

    void pushBack(int val) {
        Node* newNode = new Node(val);
        if (size == 0) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        size++;
    }

    int popFront() {
        if (size == 0) return -1;
        int val = head->val;
        Node* temp = head;
        head = head->next;
        if (head) head->prev = nullptr;
        else tail = nullptr;
        delete temp;
        size--;
        return val;
    }

    int popMiddle() {
        if (size == 0) return -1;

        Node* middle = getMiddleNode();
        int val = middle->val;

        if (middle->prev) middle->prev->next = middle->next;
        if (middle->next) middle->next->prev = middle->prev;

        if (middle == head) head = middle->next;
        if (middle == tail) tail = middle->prev;

        delete middle;
        size--;
        return val;
    }

    int popBack() {
        if (size == 0) return -1;
        int val = tail->val;
        Node* temp = tail;
        tail = tail->prev;
        if (tail) tail->next = nullptr;
        else head = nullptr;
        delete temp;
        size--;
        return val;
    }
};
```

1721. Problem: **Swapping Nodes in a Linked List**

You are given the head of a linked list, and an integer k. Swap the values of the k-th node from the beginning and the k-th node from the end (the list is 1-indexed).

Example 1:
Input: `head = [1,2,3,4,5], k = 2`
Output: `[1,4,3,2,5]`
Explanation: Swap nodes 2 (value 2) and 4 (value 4).

Example 2:
Input: `head = [7,9,6,6,7,8,3,0,9,5], k = 5`
Output: `[7,9,6,6,8,7,3,0,9,5]`

Solution 1: Two Pointers - Find Both Nodes Then Swap

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

```cpp
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        if (!head || !head->next) return head;

        ListNode* first = head;
        ListNode* second = head;
        ListNode* fast = head;

        // Move first to k-th node from beginning
        for (int i = 1; i < k; i++) {
            first = first->next;
            fast = fast->next;
        }

        // Move fast to end, second will become k-th from end
        while (fast->next) {
            second = second->next;
            fast = fast->next;
        }

        swap(first->val, second->val);
        return head;
    }
};
```

Solution 2: Three Pass - Count Length First

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

```cpp
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        if (!head || !head->next) return head;

        // First pass: count length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Find k-th from beginning and k-th from end
        ListNode* first = head;
        for (int i = 1; i < k; i++) {
```

```
            first = first->next;
        }

        ListNode* second = head;
        for (int i = 1; i < length - k + 1; i++) {
            second = second->next;
        }

        // Swap values
        swap(first->val, second->val);

        return head;
    }
};
```

Solution 3: Using Vector/Array

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

```
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        vector<ListNode*> nodes;
        ListNode* curr = head;

        // Store all nodes in vector
        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        // Swap k-th from beginning and k-th from end
        int n = nodes.size();
        swap(nodes[k-1]->val, nodes[n-k]->val);

        return head;
    }
};
```

Solution 4: One Pass with Two Pointers (Alternative)

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

```
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        ListNode* node1 = nullptr;
        ListNode* node2 = nullptr;
        ListNode* curr = head;

        // Traverse and find both nodes
        while (curr) {
            if (node2) node2 = node2->next;

            k--;
            if (k == 0) {
                node1 = curr;
                node2 = head; // Start node2 from head when we found node1
            }
            curr = curr->next;
        }

        swap(node1->val, node2->val);
        return head;
    }
};
```

Solution 5: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        ListNode* first = nullptr;
        ListNode* second = nullptr;
        int count = 0;

        findNodes(head, k, count, first, second);

        if (first && second) {
            swap(first->val, second->val);
        }
        return head;
    }

private:
    void findNodes(ListNode* node, int k, int& count,
ListNode*& first, ListNode*& second) {
        if (!node) return;

        count++;
        if (count == k) {
            first = node;
        }

        findNodes(node->next, k, count, first, second);

        count--;
        if (count == k) {
            second = node;
        }
    }
};
```

---

solution 6: Using Stack

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

```
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        stack<ListNode*> st;
        ListNode* curr = head;
        ListNode* first = nullptr;
        int count = 0;

        // Find first node and push all nodes to stack
        while (curr) {
            count++;
            if (count == k) {
                first = curr;
            }
            st.push(curr);
            curr = curr->next;
        }

        // Pop k nodes to get k-th from end
        ListNode* second = nullptr;
        for (int i = 0; i < k; i++) {
            second = st.top();
            st.pop();
        }

        // Swap values
        swap(first->val, second->val);

        return head;
    }
};
```

2058. Problem: **Find the Minimum and Maximum Number of Nodes Between Critical Points**

A critical point in a linked list is defined as either a local maxima or a local minima. A node is a local maxima if the current node has a value strictly greater than the previous node and the next node. A node is a local minima if the current node has a value strictly less than the previous node and the next node.

Given a linked list head, return an array of length 2 containing [minDistance, maxDistance] where:
- minDistance is the minimum distance between any two distinct critical points
- maxDistance is the maximum distance between any two distinct critical points

If there are fewer than two critical points, return [-1, -1].

Example 1:
Input: `head = [3,1]`
Output: `[-1,-1]`
Explanation: There are no critical points.

Example 2:
Input: `head = [5,3,1,2,5,1,2]`
Output: `[1,3]`
Explanation: Critical points at positions 2, 4, 5. minDistance = 1 (between 4 and 5), maxDistance = 3 (between 2 and 5).

Example 3:
Input: `head = [1,3,2,2,3,2,2,2,7]`
Output: `[3,3]`
Explanation: Critical points at positions 2, 5. minDistance = 3, maxDistance = 3.

Solution 1: One Pass with Critical Points Tracking

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

```
class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode*
head) {
        if (!head || !head->next || !head->next->next) {
            return {-1, -1};
        }

        ListNode* prev = head;
        ListNode* curr = head->next;
        ListNode* next = curr->next;

        vector<int> positions;
        int pos = 1; // position of curr node

        while (next) {
            // Check if current node is critical point
            if ((curr->val > prev->val && curr->val > next->val)
||
                (curr->val < prev->val && curr->val < next->val))
{
                positions.push_back(pos);
            }

            prev = curr;
            curr = next;
            next = next->next;
            pos++;
        }

        // If less than 2 critical points
        if (positions.size() < 2) {
            return {-1, -1};
        }

        // Calculate min and max distances
        int minDist = INT_MAX;
        int maxDist = positions.back() - positions.front();

        for (int i = 1; i < positions.size(); i++) {
```

```
            minDist = min(minDist, positions[i] - positions[i - 1]);
        }

        return {minDist, maxDist};
    }
};
```

Solution 2: Store Critical Point Nodes

Time Complexity: O(n)
Space Complexity: O(k) where k is number of critical points

```
class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
        vector<ListNode*> criticalPoints;
        ListNode* prev = nullptr;
        ListNode* curr = head;

        int pos = 0;
        while (curr && curr->next) {
            if (prev && curr->next) {
                // Check for local maxima or minima
                if ((curr->val > prev->val && curr->val > curr->next->val) ||
                    (curr->val < prev->val && curr->val < curr->next->val)) {
                    criticalPoints.push_back(curr);
                }
            }
            prev = curr;
            curr = curr->next;
            pos++;
        }

        if (criticalPoints.size() < 2) {
            return {-1, -1};
        }

        int minDist = INT_MAX;
        int maxDist = 0;

        // Max distance is between first and last critical point
        ListNode* first = criticalPoints.front();
        ListNode* last = criticalPoints.back();

        // Calculate positions by traversing
        curr = head;
        int firstPos = -1, lastPos = -1;
        pos = 0;
        while (curr) {
            if (curr == first) firstPos = pos;
            if (curr == last) lastPos = pos;
            curr = curr->next;
            pos++;
        }
        maxDist = lastPos - firstPos;

        // Calculate min distance
        for (int i = 1; i < criticalPoints.size(); i++) {
            curr = head;
            int prevPos = -1, currPos = -1;
            pos = 0;
            while (curr) {
                if (curr == criticalPoints[i-1]) prevPos = pos;
                if (curr == criticalPoints[i]) currPos = pos;
                curr = curr->next;
                pos++;
            }
            minDist = min(minDist, currPos - prevPos);
        }

        return {minDist, maxDist};
    }
};
```

Solution 3: Optimized with Position Tracking

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

```
class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
        if (!head || !head->next || !head->next->next) {
            return {-1, -1};
        }

        ListNode* prev = head;
        ListNode* curr = head->next;
        ListNode* next = curr->next;

        int firstCritical = -1, lastCritical = -1;
        int prevCritical = -1;
        int minDist = INT_MAX;
        int pos = 1;

        while (next) {
            // Check if current node is critical point
            bool isCritical = (curr->val > prev->val && curr->val > next->val) ||
                              (curr->val < prev->val && curr->val < next->val);

            if (isCritical) {
                if (firstCritical == -1) {
                    firstCritical = pos;
                } else {
                    minDist = min(minDist, pos - prevCritical);
                }
                prevCritical = pos;
                lastCritical = pos;
            }

            prev = curr;
            curr = next;
            next = next->next;
            pos++;
        }

        if (firstCritical == -1 || lastCritical == firstCritical) {
            return {-1, -1};
        }

        int maxDist = lastCritical - firstCritical;
        return {minDist, maxDist};
    }
};
```

Solution 4: Using Array to Store Positions

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

```
class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
        vector<int> criticalPositions;

        if (!head || !head->next || !head->next->next) {
            return {-1, -1};
        }

        ListNode* prev = head;
        ListNode* curr = head->next;
        int pos = 1; // position of current node

        while (curr->next) {
            ListNode* next = curr->next;

            // Check for critical point
            if ((curr->val > prev->val && curr->val > next->val) ||
                (curr->val < prev->val && curr->val < next->val)) {
                criticalPositions.push_back(pos);
            }

            prev = curr;
            curr = next;
            pos++;
        }

        if (criticalPositions.size() < 2) {
            return {-1, -1};
        }

        int minDist = INT_MAX;
        for (int i = 1; i < criticalPositions.size(); i++) {
            minDist = min(minDist, criticalPositions[i] - criticalPositions[i-1]);
        }

        int maxDist = criticalPositions.back() - criticalPositions.front();

        return {minDist, maxDist};
    }
};
```

Solution 5: Early Termination

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

```
class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
        if (!head || !head->next || !head->next->next) {
            return {-1, -1};
        }

        ListNode* prev = head;
        ListNode* curr = head->next;
        ListNode* next = curr->next;

        int first = -1, last = -1;
        int minDist = INT_MAX;
        int prevPos = -1;
        int pos = 1;

        while (next) {
            // Check if current is critical point
            if ((curr->val > prev->val && curr->val > next->val) ||
                (curr->val < prev->val && curr->val < next->val)) {

                if (first == -1) {
                    first = pos;
                } else {
                    minDist = min(minDist, pos - prevPos);
                }
                prevPos = pos;
                last = pos;
            }

            // Move pointers
            prev = curr;
            curr = next;
            next = next->next;
            pos++;
        }
```

```
        if (first == last) { // Only one critical point
            return {-1, -1};
        }

        return {minDist, last - first};
    }
};
```

2074 Problem: **Reverse Nodes in Even Length Groups**

You are given the head of a linked list. The nodes in the linked list are sequentially assigned to non-empty groups whose lengths form the sequence of the natural numbers (1, 2, 3, 4, ...). The length of a group is the number of nodes assigned to it.

In every group, the nodes are reversed if the group length is even. Return the modified linked list.

Example 1:
Input: `head = [5,2,6,3,9,1,7,3,8,4]`
Output: `[5,6,2,3,9,1,4,8,3,7]`
Explanation:
- Group 1: length 1 → [5] (no reverse)
- Group 2: length 2 → [2,6] → reverse → [6,2]
- Group 3: length 3 → [3,9,1] (no reverse)
- Group 4: length 4 → [7,3,8,4] → reverse → [4,8,3,7]

Example 2:
Input: `head = [1,1,0,6]`
Output: `[1,0,1,6]`

Example 3:
Input: `head = [1,1,0,6,5]`
Output: `[1,0,1,5,6]`

Solution 1: Group Processing with Reversal

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

```
class Solution {
public:
    ListNode* reverseEvenLengthGroups(ListNode* head)
{
        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        ListNode* prevGroupEnd = dummy;
        ListNode* curr = head;
        int groupNum = 1;

        while (curr) {
            int count = 0;
            ListNode* groupStart = curr;
            ListNode* groupEnd = curr;

            // Traverse the current group
            while (count < groupNum && curr) {
                groupEnd = curr;
                curr = curr->next;
                count++;
            }

            // If group has even length, reverse it
            if (count % 2 == 0) {
                // Reverse the group from groupStart to
groupEnd
                ListNode* reversedHead =
reverseList(groupStart, groupEnd->next);
                prevGroupEnd->next = reversedHead;
                groupStart->next = curr; // Connect to next
group
                prevGroupEnd = groupStart;
            } else {
                // For odd length, just update prevGroupEnd
                prevGroupEnd = groupEnd;
            }
            groupNum++;
        }
        return dummy->next;
    }

private:
    // Reverse linked list from start to end (exclusive)
    ListNode* reverseList(ListNode* start, ListNode* end) {
        ListNode* prev = end;
        ListNode* curr = start;

        while (curr != end) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

Solution 2: Count Total Length First

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

```
class Solution {
public:
    ListNode* reverseEvenLengthGroups(ListNode* head)
{
        // First, count total length
        int totalLength = 0;
        ListNode* curr = head;
        while (curr) {
            totalLength++;
            curr = curr->next;
        }

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;

        int groupNum = 1;
        int processed = 0;

        while (processed < totalLength) {
            int groupSize = min(groupNum, totalLength -
processed);

            if (groupSize % 2 == 0) {
                // Reverse this even-length group
                ListNode* groupStart = prev->next;
                ListNode* groupEnd = groupStart;

                // Move to the end of current group
                for (int i = 1; i < groupSize; i++) {
                    groupEnd = groupEnd->next;
                }

                ListNode* nextGroup = groupEnd->next;
                ListNode* reversed = reverseList(groupStart,
nextGroup);

                prev->next = reversed;
                groupStart->next = nextGroup;
                prev = groupStart;
            } else {
                // Just move pointers for odd-length group
                for (int i = 0; i < groupSize; i++) {
                    prev = prev->next;
                }
            }

            processed += groupSize;
            groupNum++;
        }

        return dummy->next;
    }

private:
    ListNode* reverseList(ListNode* start, ListNode* end) {
        ListNode* prev = nullptr;
        ListNode* curr = start;

        while (curr != end) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 3: Recursive Group Processing

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* reverseEvenLengthGroups(ListNode* head)
{
        return processGroup(head, 1);
    }

private:
    ListNode* processGroup(ListNode* head, int
groupNum) {
        if (!head) return nullptr;

        int count = 0;
        ListNode* curr = head;
        ListNode* groupEnd = head;

        // Count nodes in current group
        while (count < groupNum && curr) {
            groupEnd = curr;
            curr = curr->next;
            count++;
        }

        ListNode* nextGroup = processGroup(curr,
groupNum + 1);

        if (count % 2 == 0) {
            // Reverse current group
            ListNode* reversed = reverseList(head, curr);
            groupEnd->next = nextGroup;
            return reversed;
        } else {
            // Keep current group as is
            groupEnd->next = nextGroup;
            return head;
        }
    }

    ListNode* reverseList(ListNode* start, ListNode* end) {
        ListNode* prev = nullptr;
        ListNode* curr = start;

        while (curr != end) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
```

```
    }
};
```

Solution 4: Using Stack for Reversal

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

```
class Solution {
public:
    ListNode* reverseEvenLengthGroups(ListNode* head)
{
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;

        int groupNum = 1;

        while (head) {
            stack<ListNode*> st;
            ListNode* groupStart = head;
            int count = 0;

            // Push current group to stack
            while (count < groupNum && head) {
                st.push(head);
                head = head->next;
                count++;
            }

            if (count % 2 == 0) {
                // Pop from stack to reverse
                ListNode* curr = prev;
                while (!st.empty()) {
                    curr->next = st.top();
                    st.pop();
                    curr = curr->next;
                }
                curr->next = head; // Connect to next group
                prev = curr;
            } else {
                // Move prev to end of current group
                while (prev->next != head) {
                    prev = prev->next;
                }
            }

            groupNum++;
        }

        return dummy->next;
    }
};
```

Solution 5: In-place Group Processing

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

```
class Solution {
public:
    ListNode* reverseEvenLengthGroups(ListNode* head)
{
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prevGroupEnd = dummy;

        int groupSize = 1;

        while (head) {
            // Count actual nodes in current group
            int count = 0;
            ListNode* groupStart = head;
            ListNode* groupEnd = head;

            while (count < groupSize && head) {
                groupEnd = head;
                head = head->next;
                count++;
            }

            if (count % 2 == 0) {
                // Reverse the group in-place
                ListNode* prev = nullptr;
                ListNode* curr = groupStart;

                for (int i = 0; i < count; i++) {
                    ListNode* next = curr->next;
                    curr->next = prev;
                    prev = curr;
                    curr = next;
                }

                prevGroupEnd->next = prev;
                groupStart->next = head;
                prevGroupEnd = groupStart;
            } else {
                prevGroupEnd = groupEnd;
            }

            groupSize++;
        }

        return dummy->next;
    }
};
```

2095. Problem: **Delete the Middle Node of a Linked List**

You are given the head of a linked list. Delete the middle node, and return the head of the modified linked list.

The middle node of a linked list of size n is the ⌊n/2⌋-th node from the start using 0-based indexing.

If there's exactly one node, return null.

Example 1:
Input: `head = [1,3,4,7,1,2,6]`
Output: `[1,3,4,1,2,6]`
Explanation: The middle node with value 7 is deleted.

Example 2:
Input: `head = [1,2,3,4]`
Output: `[1,2,4]`
Explanation: The middle node with value 3 is deleted.

Example 3:
Input: `head = [2,1]`
Output: `[2]`
Explanation: The middle node with value 1 is deleted.

Solution 1: Slow and Fast Pointers with Dummy Node

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

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        ListNode* slow = dummy;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Slow is now the node before middle
        slow->next = slow->next->next;

        return dummy->next;
    }
};
```

Solution 2: Slow and Fast Pointers without Dummy

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

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;

        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }

        // Prev is the node before middle
        prev->next = slow->next;
        return head;
    }
};
```

Solution 3: Two Pass - Count then Delete

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

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        // First pass: count nodes
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Second pass: delete middle
        int middle = length / 2;
        curr = head;
        for (int i = 0; i < middle - 1; i++) {
            curr = curr->next;
        }

        curr->next = curr->next->next;
        return head;
    }
};
```

Solution 4: Using Vector/Array

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

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        vector<ListNode*> nodes;
        ListNode* curr = head;

        // Store all nodes in vector
        while (curr) {
            nodes.push_back(curr);
            curr = curr->next;
        }

        int middle = nodes.size() / 2;

        // If middle is not the last node
        if (middle < nodes.size() - 1) {
            nodes[middle - 1]->next = nodes[middle + 1];
        } else {
            nodes[middle - 1]->next = nullptr;
        }
        return head;
    }
};
```

Solution 5: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        int length = getLength(head);
        return deleteMiddleHelper(head, length, 0);
    }
private:
    int getLength(ListNode* head) {
        if (!head) return 0;
        return 1 + getLength(head->next);
    }

    ListNode* deleteMiddleHelper(ListNode* node, int
length, int index) {
        if (!node) return nullptr;

        if (index == length / 2 - 1) {
            node->next = node->next->next;
            return node;
        }
        node->next = deleteMiddleHelper(node->next, length, index + 1);
        return node;
    }
};
```

Solution 6: One Pass with Previous Pointer

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

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        ListNode* prev = nullptr;
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }

        // Delete middle node
        if (prev) {
            prev->next = slow->next;
        }
        return head;
    }
};
```

Solution 7: Modified Slow-Fast for Even Case

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

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;
        if (!head->next->next) {
            head->next = nullptr;
            return head;
        }
        ListNode* slow = head;
        ListNode* fast = head->next->next;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        slow->next = slow->next->next;
        return head;
    }
};
```

2130. Problem: **Maximum Twin Sum of a Linked List**

In a linked list of size n, where n is even, the i-th node (0-indexed) has a twin at (n-1-i)-th node. The twin sum is the sum of a node and its twin.

Return the maximum twin sum of the linked list.

Example 1:
Input: `head = [5,4,2,1]`
Output: `6`
Explanation: Nodes 0 and 3 are twins (5 + 1 = 6), Nodes 1 and 2 are twins (4 + 2 = 6). Max twin sum is 6.

Example 2:
Input: `head = [4,2,2,3]`
Output: `7`
Explanation: Nodes 0 and 3 are twins (4 + 3 = 7), Nodes 1 and 2 are twins (2 + 2 = 4). Max twin sum is 7.

Example 3:
Input: `head = [1,100000]`
Output: `100001`

Solution 1: Reverse Second Half + Two Pointers

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

```
class Solution {
public:
    int pairSum(ListNode* head) {
        if (!head) return 0;

        // Step 1: Find middle using slow and fast pointers
        ListNode* slow = head;
        ListNode* fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Step 2: Reverse second half
        ListNode* secondHalf = reverseList(slow);
        ListNode* firstHalf = head;

        // Step 3: Calculate twin sums
        int maxSum = 0;
        while (secondHalf) {
            maxSum = max(maxSum, firstHalf->val +
secondHalf->val);
            firstHalf = firstHalf->next;
            secondHalf = secondHalf->next;
        }
        return maxSum;
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 2: Using Stack

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

```
class Solution {
public:
    int pairSum(ListNode* head) {
        stack<int> st;
        ListNode* slow = head;
        ListNode* fast = head;

        // Push first half to stack
        while (fast && fast->next) {
            st.push(slow->val);
            slow = slow->next;
            fast = fast->next->next;
        }

        // Now slow is at second half, calculate twin sums
        int maxSum = 0;
        while (slow) {
            maxSum = max(maxSum, st.top() + slow->val);
            st.pop();
            slow = slow->next;
        }
        return maxSum;
    }
};
```

Solution 3: Vector/Array Approach

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

```cpp
class Solution {
public:
    int pairSum(ListNode* head) {
        vector<int> values;
        ListNode* curr = head;

        // Store all values in vector
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        int maxSum = 0;
        int n = values.size();

        // Calculate twin sums
        for (int i = 0; i < n / 2; i++) {
            int twinSum = values[i] + values[n - 1 - i];
            maxSum = max(maxSum, twinSum);
        }
        return maxSum;
    }
};
```

Solution 4: Recursive with Global Variables

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```cpp
class Solution {
private:
    ListNode* front;
    int maxSum;

public:
    int pairSum(ListNode* head) {
        front = head;
        maxSum = 0;
        traverse(head);
        return maxSum;
    }

private:
    void traverse(ListNode* node) {
        if (!node) return;

        traverse(node->next);

        // We're at the back half now, front is at front half
        if (front) {
            maxSum = max(maxSum, front->val + node->val);
            front = front->next;
        }
    }
};
```

Solution 5: Two Pointers with Length Calculation

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

```cpp
class Solution {
public:
    int pairSum(ListNode* head) {
        // First pass: count length
        int length = 0;
        ListNode* curr = head;
        while (curr) {
            length++;
            curr = curr->next;
        }

        // Find middle node
        ListNode* middle = head;
        for (int i = 0; i < length / 2; i++) {
            middle = middle->next;
        }

        // Reverse second half
        ListNode* secondHalf = reverseList(middle);
        ListNode* firstHalf = head;

        // Calculate max twin sum
        int maxSum = 0;
        for (int i = 0; i < length / 2; i++) {
            maxSum = max(maxSum, firstHalf->val +
secondHalf->val);
            firstHalf = firstHalf->next;
            secondHalf = secondHalf->next;
        }

        return maxSum;
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 6: Using Deque

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

```cpp
class Solution {
public:
    int pairSum(ListNode* head) {
        deque<int> dq;
        ListNode* curr = head;

        // Add all values to deque
        while (curr) {
            dq.push_back(curr->val);
            curr = curr->next;
        }
        int maxSum = 0;

        // Calculate twin sums from both ends
        while (!dq.empty()) {
            int twinSum = dq.front() + dq.back();
            maxSum = max(maxSum, twinSum);
            dq.pop_front();
            dq.pop_back();
        }
        return maxSum;
    }
};
```

2181. Problem: **Merge Nodes in Between Zeros**

You are given the head of a linked list, which contains a series of integers separated by 0's. The beginning and end of the linked list will have Node.val == 0.

For every two consecutive 0's, merge all the nodes lying between them into a single node whose value is the sum of all the merged nodes. The modified list should not contain any 0's.

Return the head of the modified linked list.

Example 1:
Input: `head = [0,3,1,0,4,5,2,0]`
Output: `[4,11]`
Explanation:
- Between first and second 0: 3 + 1 = 4
- Between second and third 0: 4 + 5 + 2 = 11

Example 2:
Input: `head = [0,1,0,3,0,2,2,0]`
Output: `[1,3,4]`
Explanation:
- Between first and second 0: 1 = 1
- Between second and third 0: 3 = 3
- Between third and fourth 0: 2 + 2 = 4

Solution 1: One Pass with Running Sum

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

```cpp
class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;
        ListNode* curr = head->next; // Skip first 0
        int sum = 0;

        while (curr) {
            if (curr->val == 0) {
                // Create new node with sum and reset
                tail->next = new ListNode(sum);
                tail = tail->next;
                sum = 0;
            } else {
                sum += curr->val;
            }
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

Solution 2: In-place Modification

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

```cpp
class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        ListNode* prev = head;
        ListNode* curr = head->next;
        int sum = 0;

        while (curr) {
            if (curr->val == 0) {
                // Create merged node
                prev->next = new ListNode(sum);
                prev = prev->next;
                sum = 0;
            } else {
                sum += curr->val;
            }
            curr = curr->next;
        }

        return head->next; // Skip the original first 0
    }
};
```

Solution 3: Two Pointers Approach

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

```
class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        ListNode* result = dummy;
        ListNode* start = head->next; // First non-zero after
initial 0

        while (start) {
            ListNode* end = start;
            int sum = 0;

            // Traverse until we hit next 0
            while (end && end->val != 0) {
                sum += end->val;
                end = end->next;
            }

            // Add merged node to result
            result->next = new ListNode(sum);
            result = result->next;

            // Move to next segment
            start = end ? end->next : nullptr;
        }

        return dummy->next;
    }
};
```

Solution 4: Using Vector/Array

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

```
class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        vector<int> sums;
        ListNode* curr = head;
        int sum = 0;

        while (curr) {
            if (curr->val == 0 && sum > 0) {
                sums.push_back(sum);
                sum = 0;
            } else {
                sum += curr->val;
            }
            curr = curr->next;
        }

        // Build new linked list
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;

        for (int val : sums) {
            tail->next = new ListNode(val);
            tail = tail->next;
        }
        return dummy->next;
    }
};
```

Solution 5: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        if (!head || !head->next) return nullptr;

        ListNode* curr = head->next; // Skip current 0
        int sum = 0;

        // Calculate sum until next 0
        while (curr && curr->val != 0) {
            sum += curr->val;
            curr = curr->next;
        }

        // Create merged node and recursively process rest
        ListNode* merged = new ListNode(sum);
        merged->next = mergeNodes(curr);

        return merged;
    }
};
```

Solution 6: Iterative with Previous Zero Tracking

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

```
class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        ListNode* prevZero = head;
        ListNode* curr = head->next;

        while (curr) {
            if (curr->val == 0) {
                prevZero = curr;
                curr = curr->next;
            } else {
                // Add current value to previous zero node
                prevZero->val += curr->val;
                // Remove current node
                prevZero->next = curr->next;
                curr = curr->next;
            }
        }

        // Now remove all zero nodes except the first one
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;
        curr = head;

        while (curr) {
            if (curr->val == 0 && curr != head) {
                // Remove zero node
                prev->next = curr->next;
                curr = curr->next;
            } else {
                prev = curr;
                curr = curr->next;
            }
        }
        return head->val == 0 ? head->next : head;
    }
};
```

2289.Problem: Steps to Make Array Non-decreasing

You are given a 0-indexed integer array nums. In one step, remove all elements nums[i] where nums[i] > nums[i + 1] for all 0 <= i < nums.length - 1.

Return the number of steps performed until nums becomes non-decreasing.

Example 1:
Input: `nums = [5,3,4,4,7,3,6,11,8,5,11]`
Output: `3`
Explanation: The following are the steps performed:
- Step 1: [5,3,4,4,7,3,6,11,8,5,11] → [5,4,4,7,6,11,11]
- Step 2: [5,4,4,7,6,11,11] → [5,4,7,11,11]
- Step 3: [5,4,7,11,11] → [5,7,11,11]
Now nums is non-decreasing, so the answer is 3.

Example 2:
Input: `nums = [4,5,7,7,13]`
Output: `0`
Explanation: nums is already non-decreasing.

Example 3:
Input: `nums = [1,3,2]`
Output: `1`
Explanation: Step 1: [1,3,2] → [1,3]

Solution 1: Monotonic Stack Approach

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

```
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        int n = nums.size();
        vector<int> steps(n, 0);
        stack<int> st;

        int maxSteps = 0;

        for (int i = n - 1; i >= 0; i--) {
            // Remove elements that are smaller than current
            while (!st.empty() && nums[i] > nums[st.top()]) {
                steps[i] = max(steps[i] + 1, steps[st.top()]);
                st.pop();
            }
            st.push(i);
            maxSteps = max(maxSteps, steps[i]);
        }

        return maxSteps;
    }
};
```

Solution 2: Stack with Pair (Value, Steps)

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

```
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        stack<pair<int, int>> st; // pair: {value, steps to
remove this element}
        int maxSteps = 0;

        for (int i = nums.size() - 1; i >= 0; i--) {
            int steps = 0;

            // Remove elements that current element can "eat"
            while (!st.empty() && nums[i] > st.top().first) {
                steps = max(steps + 1, st.top().second);
                st.pop();
            }

            maxSteps = max(maxSteps, steps);
            st.push({nums[i], steps});
        }

        return maxSteps;
    }
};
```

Solution 3: Simulation (Brute Force)

Time Complexity: O(n²)
Space Complexity: O(n)

```
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        vector<int> arr = nums;
        int steps = 0;

        while (true) {
            vector<int> next;
            bool changed = false;

            next.push_back(arr[0]);
            for (int i = 1; i < arr.size(); i++) {
                if (arr[i] >= arr[i - 1]) {
                    next.push_back(arr[i]);
                } else {
                    changed = true;
                }
            }

            if (!changed) break;
            steps++;
            arr = next;
        }

        return steps;
    }
};
```

Solution 4: Linked List Simulation

Time Complexity: O(n²)
Space Complexity: O(n)

```
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        list<int> lst(nums.begin(), nums.end());
        int steps = 0;

        while (true) {
            bool changed = false;
            auto it = lst.begin();
            auto nextIt = it;
            ++nextIt;

            while (nextIt != lst.end()) {
                if (*it > *nextIt) {
                    // Remove the smaller element
                    nextIt = lst.erase(nextIt);
                    changed = true;
                } else {
                    ++it;
                    ++nextIt;
                }
            }

            if (!changed) break;
            steps++;
        }

        return steps;
    }
};
```

Solution 5: Dynamic Programming with Stack

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

```
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        int n = nums.size();
        stack<int> st;
        vector<int> dp(n, 0); // dp[i] = steps needed to
remove nums[i]
        int maxSteps = 0;

        for (int i = 0; i < n; i++) {
            int currentSteps = 0;

            // Find how many elements current can remove
            while (!st.empty() && nums[st.top()] <= nums[i]) {
                currentSteps = max(currentSteps, dp[st.top()]);
                st.pop();
            }

            // If there's a larger element to the left, it can
remove current
            if (!st.empty()) {
                dp[i] = currentSteps + 1;
                maxSteps = max(maxSteps, dp[i]);
            }

            st.push(i);
        }

        return maxSteps;
    }
};
```

Solution 6: Two Pass Approach

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

```
class Solution {
public:
    int totalSteps(vector<int>& nums) {
        int n = nums.size();
        vector<int> nextGreater(n, n);
        stack<int> st;

        // Find next greater element for each position
        for (int i = 0; i < n; i++) {
            while (!st.empty() && nums[st.top()] <= nums[i]) {
                nextGreater[st.top()] = i;
                st.pop();
            }
            st.push(i);
        }

        vector<int> dp(n, 0);
        int maxSteps = 0;

        // Calculate steps from right to left
        for (int i = n - 1; i >= 0; i--) {
            if (nextGreater[i] < n) {
                dp[i] = max(dp[i], 1 + dp[nextGreater[i]]);
            }
            maxSteps = max(maxSteps, dp[i]);
        }

        return maxSteps;
    }
};
```

2487. Problem: **Remove Nodes From Linked List**

You are given the head of a linked list. Remove every node which has a node with a greater value anywhere to the right side of it. Return the head of the modified linked list.

Example 1:
Input: `head = [5,2,13,3,8]`
Output: `[13,8]`
Explanation: The nodes that should be removed are 5, 2, and 3.
- Node 13 is to the right of node 5.
- Node 13 is to the right of node 2.
- Node 8 is to the right of node 3.

Example 2:
Input: `head = [1,1,1,1]`
Output: `[1,1,1,1]`
Explanation: Every node has value 1, so no nodes are removed.

Solution 1: Reverse, Filter, Reverse Back

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

```
class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        // Step 1: Reverse the linked list
        head = reverseList(head);

        // Step 2: Remove nodes that are smaller than max
seen so far
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* curr = head;
        int maxVal = head->val;

        while (curr && curr->next) {
            if (curr->next->val < maxVal) {
                // Remove the node
                curr->next = curr->next->next;
            } else {
                // Update max and move forward
                maxVal = max(maxVal, curr->next->val);
                curr = curr->next;
            }
        }

        // Step 3: Reverse back
        return reverseList(dummy->next);
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 2: Monotonic Stack

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

```
class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        stack<ListNode*> st;
        ListNode* curr = head;

        // Push all nodes to stack
        while (curr) {
```

```
        // Remove nodes from stack that are smaller than current
        while (!st.empty() && st.top()->val < curr->val) {
            st.pop();
        }
        st.push(curr);
        curr = curr->next;
    }

    // Build result from stack (in reverse order)
    ListNode* next = nullptr;
    while (!st.empty()) {
        ListNode* node = st.top();
        st.pop();
        node->next = next;
        next = node;
    }

    return next;
  }
};
```

Solution 3: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        if (!head || !head->next) return head;

        // Recursively process the rest
        head->next = removeNodes(head->next);

        // If current node is smaller than next, remove current
        if (head->next && head->val < head->next->val) {
            return head->next;
        }

        return head;
    }
};
```

Solution 4: Using Vector/Array

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

```
class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        vector<int> values;
        ListNode* curr = head;

        // Convert to array
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        // Find nodes to keep using monotonic stack from right
        stack<int> st;
        for (int i = values.size() - 1; i >= 0; i--) {
            if (st.empty() || values[i] >= st.top()) {
                st.push(values[i]);
            }
        }

        // Build new linked list
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;

        while (!st.empty()) {
            tail->next = new ListNode(st.top());
            tail = tail->next;
            st.pop();
        }

        return dummy->next;
    }
};
```

Solution 5: One Pass with Max Tracking

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

```
class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        // Reverse the list first
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        // Now traverse reversed list and keep only nodes >= current max
        ListNode* newHead = nullptr;
        int maxVal = INT_MIN;
        curr = prev;

        while (curr) {
            if (curr->val >= maxVal) {
                maxVal = curr->val;
                ListNode* newNode = new ListNode(curr->val);
                newNode->next = newHead;
                newHead = newNode;
            }
            curr = curr->next;
        }
        return newHead;
    }
};
```

Solution 6: In-place Modification with Two Pointers

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

```
class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        if (!head || !head->next) return head;

        // Reverse the list
        ListNode* reversed = reverseList(head);

        // Filter nodes
        ListNode* curr = reversed;
        int maxVal = curr->val;

        while (curr->next) {
            if (curr->next->val < maxVal) {
                curr->next = curr->next->next;
            } else {
                maxVal = max(maxVal, curr->next->val);
                curr = curr->next;
            }
        }

        // Reverse back
        return reverseList(reversed);
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

2807. Problem: **Insert Greatest Common Divisors in Linked List**

Given the head of a linked list head, in which each node contains an integer value.

Between every pair of adjacent nodes, insert a new node with a value equal to the greatest common divisor of them.

Return the linked list after insertion.

The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers.

Example 1:
Input: `head = [18,6,10,3]`
Output: `[18,6,6,2,10,1,3]`
Explanation:
- GCD(18,6) = 6 → Insert 6 between 18 and 6
- GCD(6,10) = 2 → Insert 2 between 6 and 10
- GCD(10,3) = 1 → Insert 1 between 10 and 3

Example 2:
Input: `head = [7]`
Output: `[7]`
Explanation: Only one node, so no insertion.

Solution 1: Iterative with GCD Calculation

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

```
class Solution {
public:
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* curr = head;

        while (curr && curr->next) {
            int gcdVal = gcd(curr->val, curr->next->val);
            ListNode* newNode = new ListNode(gcdVal);

            // Insert new node between curr and curr->next
            newNode->next = curr->next;
            curr->next = newNode;

            // Move to the next original node
            curr = newNode->next;
        }

        return head;
    }
```

```
private:
    int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
};
```

Solution 2: Using Built-in GCD

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

```
class Solution {
public:
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* curr = head;

        while (curr && curr->next) {
            int gcdVal = __gcd(curr->val, curr->next->val); // Built-in GCD
            ListNode* newNode = new ListNode(gcdVal);

            newNode->next = curr->next;
            curr->next = newNode;
            curr = newNode->next;
        }

        return head;
    }
};
```

Solution 3: Recursive Approach

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if (!head || !head->next) return head;

        // Insert GCD between head and head->next
        int gcdVal = gcd(head->val, head->next->val);
        ListNode* newNode = new ListNode(gcdVal);

        newNode->next = head->next;
        head->next = newNode;

        // Recursively process the rest
        insertGreatestCommonDivisors(newNode->next);

        return head;
    }

private:
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
};
```

Solution 4: Two Pointers with Previous

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

```
class Solution {
public:
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if (!head || !head->next) return head;

        ListNode* prev = head;
        ListNode* curr = head->next;

        while (curr) {
            int gcdVal = gcd(prev->val, curr->val);
            ListNode* newNode = new ListNode(gcdVal);

            prev->next = newNode;
            newNode->next = curr;

            prev = curr;
            curr = curr->next;
        }

        return head;
    }

private:
    int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
};
```

Solution 5: Using Vector/Array

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

```
class Solution {
public:
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if (!head || !head->next) return head;

        vector<int> values;
        ListNode* curr = head;

        // Store all values
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        // Build new list with GCDs inserted
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;

        for (int i = 0; i < values.size(); i++) {
            // Add current node
            tail->next = new ListNode(values[i]);
            tail = tail->next;

            // Add GCD if not last node
            if (i < values.size() - 1) {
                int gcdVal = gcd(values[i], values[i + 1]);
                tail->next = new ListNode(gcdVal);
                tail = tail->next;
            }
        }

        return dummy->next;
    }

private:
    int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
};
```

Solution 6: In-place with While Loop

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

```
class Solution {
public:
    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        ListNode* curr = head;

        while (curr && curr->next) {
            // Calculate GCD
            int a = curr->val;
            int b = curr->next->val;
            int gcdVal = computeGCD(a, b);

            // Create and insert new node
            ListNode* gcdNode = new ListNode(gcdVal);
            gcdNode->next = curr->next;
            curr->next = gcdNode;

            // Move to next original node
            curr = gcdNode->next;
        }

        return head;
    }

private:
    int computeGCD(int a, int b) {
        if (b == 0) return a;
        return computeGCD(b, a % b);
    }
};
```

2816. Problem: **Double a Number Represented as a Linked List**

You are given the head of a non-empty linked list representing a non-negative integer without leading zeroes. Double the number and return the head of the modified linked list.

Example 1:
Input: `head = [1,8,9]`
Output: `[3,7,8]`
Explanation: 189 × 2 = 378

Example 2:
Input: `head = [9,9,9]`
Output: `[1,9,9,8]`
Explanation: 999 × 2 = 1998

Solution 1: Reverse, Double, Reverse Back

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

```
class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        // Reverse the linked list
        head = reverseList(head);

        ListNode* curr = head;
        ListNode* prev = nullptr;
        int carry = 0;
```

```
        // Double each digit and handle carry
        while (curr) {
            int doubled = curr->val * 2 + carry;
            curr->val = doubled % 10;
            carry = doubled / 10;
            prev = curr;
            curr = curr->next;
        }

        // If there's remaining carry, add new node
        if (carry > 0) {
            prev->next = new ListNode(carry);
        }

        // Reverse back
        return reverseList(head);
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 2: Recursive with Carry

Time Complexity: O(n)
Space Complexity: O(n) - recursion stack

```
class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        int carry = doubleHelper(head);
        if (carry > 0) {
            ListNode* newHead = new ListNode(carry);
            newHead->next = head;
            return newHead;
        }
        return head;
    }

private:
    int doubleHelper(ListNode* node) {
        if (!node) return 0;
        int doubled = node->val * 2 +
doubleHelper(node->next);
        node->val = doubled % 10;
        return doubled / 10;
    }
};
```

Solution 3: Using Stack

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

```
class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        stack<ListNode*> st;
        ListNode* curr = head;

        // Push all nodes to stack
        while (curr) {
            st.push(curr);
            curr = curr->next;
        }

        int carry = 0;
        ListNode* newHead = nullptr;

        // Process from least significant digit
        while (!st.empty()) {
            ListNode* node = st.top();
            st.pop();

            int doubled = node->val * 2 + carry;
            node->val = doubled % 10;
            carry = doubled / 10;

            // Build result in correct order
            node->next = newHead;
            newHead = node;
        }

        // Handle remaining carry
        if (carry > 0) {
            ListNode* carryNode = new ListNode(carry);
            carryNode->next = newHead;
            newHead = carryNode;
        }

        return newHead;
    }
};
```

Solution 4: Two Pass with Carry Forward

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

```
class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        // First pass: reverse the list
        ListNode* reversed = reverseList(head);

        // Second pass: double with carry
        ListNode* curr = reversed;
        int carry = 0;
        ListNode* prev = nullptr;

        while (curr) {
            int sum = curr->val * 2 + carry;
            curr->val = sum % 10;
            carry = sum / 10;
            prev = curr;
            curr = curr->next;
        }

        // Handle final carry
        if (carry > 0) {
            prev->next = new ListNode(carry);
        }

        // Reverse back
        return reverseList(reversed);
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
};
```

Solution 5: In-place Modification with Previous Pointer

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

```
class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        // If first digit >= 5, we'll need a new head
        if (head->val >= 5) {
            ListNode* newHead = new ListNode(0);
            newHead->next = head;
            head = newHead;
        }

        ListNode* curr = head;

        while (curr) {
            // Double current digit and add carry from next
            curr->val = curr->val * 2;
            if (curr->next && curr->next->val >= 5) {
                curr->val += 1;
            }
            curr->val %= 10;
            curr = curr->next;
        }
        return head;
    }
};
```

Solution 6: Vector/Array Approach

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

```
class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        vector<int> digits;
        ListNode* curr = head;

        // Convert to array
        while (curr) {
            digits.push_back(curr->val);
            curr = curr->next;
        }
        // Double the number with carry
        int carry = 0;
        for (int i = digits.size() - 1; i >= 0; i--) {
            int doubled = digits[i] * 2 + carry;
            digits[i] = doubled % 10;
            carry = doubled / 10;
        }
        // Build new linked list
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;
        // Add carry as first digit if needed
        if (carry > 0) {
            tail->next = new ListNode(carry);
            tail = tail->next;
        }
        // Add remaining digits
        for (int digit : digits) {
            tail->next = new ListNode(digit);
            tail = tail->next;
        }
        return dummy->next;
    }
};
```

3217. Problem: **Delete Nodes From Linked List Present in Array**

You are given an array of integers `nums` and the head of a linked list. Return the head of the modified linked list after removing all nodes from the linked list that have a value present in `nums`.

Example 1:
Input: `nums = [1,2,3], head = [1,2,3,4,5]`
Output: `[4,5]`
Explanation: Remove nodes with values 1, 2, 3.

Example 2:
Input: `nums = [1], head = [1,2,1,2,1,2]`
Output: `[2,2,2]`
Explanation: Remove all nodes with value 1.

Example 3:
Input: `nums = [5], head = [1,2,3,4]`
Output: `[1,2,3,4]`
Explanation: No nodes to remove.

Solution 1: HashSet with Dummy Node

Time Complexity: O(n + m) where n is list length, m is nums length
Space Complexity: O(m)

```
class Solution {
public:
   ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        unordered_set<int> toRemove(nums.begin(), nums.end());

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* curr = dummy;

        while (curr->next) {
            if (toRemove.count(curr->next->val)) {
                // Remove the node
                ListNode* temp = curr->next;
                curr->next = curr->next->next;
                delete temp;
            } else {
                curr = curr->next;
            }
        }

        return dummy->next;
    }
};
```

Solution 2: In-place Removal without Dummy

Time Complexity: O(n + m)
Space Complexity: O(m)

```
class Solution {
public:
   ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        unordered_set<int> toRemove(nums.begin(), nums.end());

        // Remove leading nodes that need to be deleted
        while (head && toRemove.count(head->val)) {
            ListNode* temp = head;
            head = head->next;
            delete temp;
        }

        if (!head) return nullptr;

        // Remove non-leading nodes
        ListNode* curr = head;
        while (curr->next) {
            if (toRemove.count(curr->next->val)) {
                ListNode* temp = curr->next;
                curr->next = curr->next->next;
                delete temp;
            } else {
                curr = curr->next;
            }
        }
        return head;
    }
};
```

Solution 3: Using Array for Small Values

Time Complexity: O(n + m)
Space Complexity: O(1001) = O(1) since values are 0 <= Node.val <= 1000

```
class Solution {
public:
   ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        vector<bool> toRemove(1001, false);
        for (int num : nums) {
            toRemove[num] = true;
        }

        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* curr = dummy;

        while (curr->next) {
            if (curr->next->val <= 1000 && toRemove[curr->next->val]) {
                ListNode* temp = curr->next;
                curr->next = curr->next->next;
                delete temp;
            } else {
                curr = curr->next;
            }
        }

        return dummy->next;
    }
};
```

Solution 4: Recursive Approach

Time Complexity: O(n + m)
Space Complexity: O(n + m) - recursion stack + hash set

```
class Solution {
public:
   ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        unordered_set<int> toRemove(nums.begin(), nums.end());
        return removeNodes(head, toRemove);
    }

private:
   ListNode* removeNodes(ListNode* node, unordered_set<int>& toRemove) {
        if (!node) return nullptr;

        if (toRemove.count(node->val)) {
            ListNode* next = removeNodes(node->next, toRemove);
            delete node;
            return next;
        } else {
            node->next = removeNodes(node->next, toRemove);
            return node;
        }
    }
};
```

Solution 5: Two Pointers with Previous

Time Complexity: O(n + m)
Space Complexity: O(m)

```
class Solution {
public:
   ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        unordered_set<int> toRemove(nums.begin(), nums.end());

        ListNode* prev = nullptr;
        ListNode* curr = head;

        while (curr) {
            if (toRemove.count(curr->val)) {
                if (prev) {
                    prev->next = curr->next;
                } else {
                    head = curr->next;
                }
                ListNode* temp = curr;
                curr = curr->next;
                delete temp;
            } else {
                prev = curr;
                curr = curr->next;
            }
        }

        return head;
    }
};
```

Solution 6: Using Vector and Rebuild

Time Complexity: O(n + m)
Space Complexity: O(n + m)

```
class Solution {
public:
   ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        unordered_set<int> toRemove(nums.begin(), nums.end());
        vector<int> remaining;
        ListNode* curr = head;

        // Collect nodes to keep
        while (curr) {
            if (!toRemove.count(curr->val)) {
                remaining.push_back(curr->val);
            }
            curr = curr->next;
        }

        // Build new linked list
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;

        for (int val : remaining) {
            tail->next = new ListNode(val);
            tail = tail->next;
        }

        return dummy->next;
    }
};
```