Linked List & Integer Problems Documentation

1. Reverse Integer (LeetCode #7)

Problem Statement

Given a signed 32-bit integer `x`, return `x` with its digits reversed. If reversing `x` causes the value to go outside the signed 32-bit integer range [-2^{31}, $2^{31} - 1$], then return 0. Assume the environment does not allow storing 64-bit integers.

Examples:

```
Input: x = 123     → Output: 321
Input: x = -123    → Output: -321
Input: x = 120     → Output: 21
```

Explanation with Cases

Case 1: Positive numbers

```
123 → 321
Digits extracted: 3, 2, 1
```

Case 2: Negative numbers

```
-123 → -321
Digits extracted: -3, -2, -1 (handled by modulo in programming)
```

Case 3: Numbers ending with zero

```
120 → 21
Leading zeros in reversed form are dropped
```

Case 4: Overflow scenarios

```
1534236469 → 0 (reverses to 9646324351 > INT_MAX)
-2147483648 → 0 (special edge case)
```

Case 5: Single digit numbers

```
5 → 5
-7 → -7
```

Approaches

Approach 1: Mathematical reversal with overflow checking

- Extract digits using modulo/division
- Check overflow before multiplication/addition
- Handle negative numbers carefully

Time Complexity: $O(\log_{10}|x|)$

Space Complexity: O(1)

Key Challenge: Handling overflow without 64-bit integers

Solution Code

```
int reverse(int x) {
    int rev = 0;
    while (x != 0) {
        int pop = x % 10;
        x /= 10;

        // Check overflow before actually doing rev * 10 + pop
        if (rev > INT_MAX/10 || (rev == INT_MAX/10 && pop > 7)) return 0;
        if (rev < INT_MIN/10 || (rev == INT_MIN/10 && pop < -8)) return 0;

        rev = rev * 10 + pop;
    }
    return rev;
}
```

Key Points:

- `INT_MAX = 2147483647` (last digit 7)
- `INT_MIN = -2147483648` (last digit -8)
- Overflow check happens BEFORE the actual operation

---

2. Merge Two Sorted Lists (LeetCode #21)

Problem Statement

Merge two sorted linked lists into one sorted list by splicing together nodes from the first two lists. Return the head of the merged linked list.

Examples:

```
List1: 1→2→4, List2: 1→3→4 → Output: 1→1→2→3→4→4
List1: [], List2: [] → Output: []
List1: [], List2: [0] → Output: [0]
```

Explanation with Cases

Case 1: Normal merge

```
List1: 1→2→4
List2: 1→3→4
Result: 1→1→2→3→4→4
```

Case 2: Empty lists

```
[] + [] = []
```

Case 3: One empty list

```
[] + [0] = [0]
[5→6] + [] = [5→6]
```

Case 4: Non-overlapping ranges

```
[1→2→3] + [4→5→6] = [1→2→3→4→5→6]
```

Case 5: Duplicate values
Handled naturally in merge logic

Approaches

Approach 1: Iterative with dummy node (Optimal)

- Create dummy node to simplify edge cases
- Use tail pointer to build result
- Compare heads, attach smaller node
- Attach remaining list at end

Time Complexity: O(n + m)

Space Complexity: O(1)

Approach 2: Recursive

- Base cases: if either list empty
- Compare heads, recursively merge rest
- Clean but uses O(n+m) stack space

Solution Code

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode dummy(0);
    ListNode* tail = &dummy;

    while (l1 && l2) {
        if (l1->val <= l2->val) {
            tail->next = l1;
            l1 = l1->next;
```

```
        } else {
            tail→next = l2;
            l2 = l2→next;
        }
        tail = tail→next;
    }

    tail→next = l1 ? l1 : l2;
    return dummy.next;
}
```

Why dummy node?

- Avoids special case for initial head
- Simplifies code significantly

---

3. Sort List (LeetCode #148)

Problem Statement

Given the head of a linked list, sort it in ascending order with O(n log n) time complexity and O(1) memory (constant space).

Examples:

```
[4→2→1→3] → [1→2→3→4]
[-1→5→3→4→0] → [-1→0→3→4→5]
[] → []
```

Explanation with Cases

Case 1: Random order

```
4→2→1→3 → 1→2→3→4
```

Case 2: Already sorted

```
1→2→3 → 1→2→3
```

Case 3: Reverse sorted

```
3→2→1 → 1→2→3
```

Case 4: Single element

```
[5] → [5]
```

Case 5: Duplicates

```
3→1→3→2 → 1→2→3→3
```

Approaches

Approach 1: Recursive Merge Sort (O(log n) stack space)

- Split list into halves using slow/fast pointer
- Recursively sort halves
- Merge sorted halves

Approach 2: Bottom-Up Merge Sort (O(1) space)

- Start with sublists of size 1
- Repeatedly merge pairs, doubling sublist size each pass
- Achieves O(1) space (no recursion stack)

Bottom-Up Merge Sort Solution

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

        // Get length
        int len = 0;
        ListNode* cur = head;
        while (cur) {
            len++;
            cur = cur->next;
        }

        ListNode dummy(0);
        dummy.next = head;

        // Bottom-up merge sort
        for (int step = 1; step < len; step <<= 1) {
            ListNode* tail = &dummy;
            cur = dummy.next;

            while (cur) {
                ListNode* left = cur;
                ListNode* right = split(left, step);
                cur = split(right, step);

                tail->next = merge(left, right);
                while (tail->next) tail = tail->next;
            }
```

```
        }
        return dummy.next;
    }

private:
    ListNode* split(ListNode* head, int n) {
        for (int i = 1; head && i < n; i++) {
            head = head->next;
        }
        if (!head) return nullptr;
        ListNode* second = head->next;
        head->next = nullptr;
        return second;
    }

    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode dummy(0);
        ListNode* tail = &dummy;
        while (l1 && l2) {
            if (l1->val <= l2->val) {
                tail->next = l1;
                l1 = l1->next;
            } else {
                tail->next = l2;
                l2 = l2->next;
            }
            tail = tail->next;
        }
        tail->next = l1 ? l1 : l2;
        return dummy.next;
    }
};
```

Summary Table

Problem	Time Complexity	Space Complexity	Key Technique
Reverse Integer	$O(\log_{10}$	x	)
Merge Two Sorted Lists	O(n+m)	O(1)	Dummy node & pointer manipulation
Sort List	O(n log n)	O(1)	Bottom-up merge sort

Common Patterns Across Problems

1. **Pointer Manipulation**: Crucial for linked list problems
2. **Divide and Conquer**: Merge sort for O(n log n) sorting
3. **Edge Case Handling**: Empty lists, single nodes, overflow

4. **Dummy Nodes**: Simplify head pointer changes
5. **In-place Operations**: Modifying structure without extra space

---

Practice Recommendations

1. **Start with**: Merge Two Sorted Lists (simplest linked list manipulation)
2. **Progress to**: Sort List (combines merging + divide-and-conquer)
3. **Master**: Reverse Integer (mathematical thinking with constraints)

Each problem builds skills for the next, particularly in handling pointers and constraints.