October 9

Problem 1: Merge Two Sorted Lists

Problem Statement:

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Link to problem:

https://leetcode.com/problems/merge-two-sorted-lists/description/

Example 1:

Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
```

```
        ListNode dNode = new ListNode(-1); // Dummy node to start the merged list
        ListNode temp = dNode, t1 = list1, t2 = list2;

        // Traverse both lists until one reaches the end
        while (t1 != null && t2 != null) {
            // If the current value of t1 is smaller, link it to the result list
            if (t1.val < t2.val) {
                temp.next = t1;
                temp = t1;  // Move the temp pointer to the next node in t1
                t1 = t1.next; // Move t1 pointer to the next node
            } else {
                temp.next = t2;
                temp = t2;  // Move the temp pointer to the next node in t2
                t2 = t2.next; // Move t2 pointer to the next node
            }
        }

        // If one list is exhausted, link the remaining elements of the other list
        if (t1 != null) {
            temp.next = t1;
        } else {
            temp.next = t2;
        }

        // Return the head of the merged list, ignoring the dummy node
        return dNode.next;
    }
}
```

Explanation:

- We create a dummy node dNode to act as the head of the merged list, allowing us to easily return the result at the end.
- temp is a pointer that tracks the current position in the merged list, starting at the dummy node.
- t1 and t2 represent pointers for traversing through list1 and list2, respectively.
- We iterate over both lists, linking the smaller of t1 or t2 to the temp node.

This ensures that the resulting list remains sorted.

- If one list becomes empty before the other, we attach the remaining portion of the other list to temp.next.
- Finally, we return the merged list starting from dNode.next (ignoring the dummy node).

Time Complexity:

- **O(n + m)**, where n and m are the lengths of the two lists. We traverse both lists only once.

Space Complexity:

- **O(1)**, since we only use a constant amount of extra space for the pointers. The space used by the output list is not counted, as it is dependent on the input

---

Problem 2: Valid Parentheses

Problem Statement:

Given a string s containing just the characters (,), {, }, [and], determine if the input string is valid. An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Link to problem:

https://leetcode.com/problems/valid-parentheses/

Example 1:

- **Input:** `s = "()"`
- **Output:** `true`

Example 2:

- **Input:** `s = "()[]{}"`
- **Output:** `true`

Example 3:

- **Input:** `s = "(]"`
- **Output:** `false`

Solution:

```
import java.util.Stack;

class Solution {
```

```
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        // Traverse through each character of the string
        for (char c : s.toCharArray()) {
            // If an opening bracket, push to the stack
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else {
                // If it's a closing bracket and the stack is empty, return false
                if (stack.isEmpty()) return false;

                // Pop the top of the stack and check if it matches the corresponding opening bracket
                char top = stack.pop();
                if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' && top != '[')) {
                    return false;
                }
            }
        }

        // If the stack is empty, all parentheses are matched, otherwise it's invalid
        return stack.isEmpty();
    }
}
```

Explanation:

- We use a stack data structure to help track opening brackets. The basic idea is:
    1. **Push** every opening bracket ((, {, [) to the stack.
    2. **Pop** the stack when encountering a closing bracket (), },]), and check whether it matches the most recent opening bracket.
    3. If it doesn't match, return false. If the stack is empty and we encounter a closing bracket, return false.
- After traversing the entire string, the stack should be empty. If there are still unmatched opening brackets in the stack, the string is invalid.

Edge Cases:

- If the string starts with a closing bracket, it will immediately return `false`.
- If there are unmatched opening brackets left in the stack, the function will return `false` as the string is not properly balanced.

Time Complexity:

- O(n), where n is the length of the string. We traverse the string once, and each stack operation (push and pop) is O(1).

Space Complexity:

- O(n) in the worst case, where all the characters are opening brackets and we store them in the stack.
