# JavaScript Sort List

## Challenge

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

### 1st Example

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

### 2nd Example

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

### 3rd Example

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

### Constraints

- `-10⁵ <= Node.val <= 10⁵`
- The number of nodes in the list is in the range `[0, 5 * 10⁴]`.

## Solution

```
const sortList = head => {
    const getMid = head => {
        let slow = head,
            fast = head.next;

        while (fast && fast.next) {
            slow = slow.next;
            fast = fast.next.next;
        }

        const M = slow.next;

        slow.next = null;

        return M;
    };

    const merge = (l1, l2) => {
        const sentinel = new ListNode(null);
        let tail = sentinel;

        while (l1 && l2) {
            if (l1.val < l2.val) {
                tail.next = l1;
                l1 = l1.next;
            } else {
                tail.next = l2;
                l2 = l2.next;
            }

            tail = tail.next;
        }
```

Solution continues on next page...

```
        tail.next = l1 || l2;

        return sentinel.next;
    };

    if (!head || !head.next) return head;

    const M     = getMid(head),
          Left  = sortList(head),
          Right = sortList(M);

    return merge(Left, Right);
};
```

## Explanation:

I've coded a function called `sortList` that implements the merge sort algorithm to sort a linked list in ascending order.

The function begins by defining an inner function called `getMid`. This function takes a linked list head as input and finds the middle node of the linked list using the two-pointer technique. It initializes a slow pointer and a fast pointer, both starting from the head. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. This continues until the fast pointer reaches the end of the list or the second-to-last node. The slow pointer will then be at the middle node. The function separates the list into two parts by setting the next pointer of the slow pointer to null and returns the middle node.

Another inner function called `merge` is defined. This function takes two sorted linked lists, `l1` and `l2`, as input and merges them into a single sorted linked list. It starts by creating a sentinel node with a null value. The function uses a tail pointer to keep track of the last node of the merged list, initially pointing to the sentinel node. It iterates through the two lists, comparing the values of the current nodes. The smaller value is appended to the tail of the merged list, and the tail pointer is updated accordingly. This process continues until one of the lists is exhausted. Then, the remaining nodes of the other list are appended to the tail. Finally, the function returns the next pointer of the sentinel node, which points to the merged list.

The main `sortList` function first checks if the input linked list is empty or has only one node. In such cases, the function immediately returns the head as it is already sorted.

If the input list has more than one node, the function proceeds to split the list into two halves. It calls the `getMid` function to find the middle node (`M`) of the list. Then, it recursively calls `sortList` on the left half (`head`) and the right half (`M`) of the list. This recursive step divides the problem into smaller subproblems, following the divide-and-conquer approach of merge sort.

Finally, the function merges the sorted left and right halves using the `merge` function. It returns the sorted linked list obtained from the merge operation.

In summary, the `sortList` function uses the merge sort algorithm to sort a linked list. It divides the list into smaller halves, sorts them individually using recursion, and then merges the sorted halves to obtain the final sorted linked list.

Author: Trevor Morin