

I've provided solutions for 8 LeetCode questions along with a solution recording video on GitHub: https://github.com/JINGJINGPAN1/cs_5008/tree/main/mock

Retrospective

1. Problem Description (2 points)

- What problem did you solve in the mock interview?
[148 Sort List](https://leetcode.com/problems/sort-list/description/)(<https://leetcode.com/problems/sort-list/description/>)
- What were the problem's requirements, constraints, and necessary data structures?
 - Requirement: Given the head of a linked list, return the list after sorting it in ascending order.
 - Constraint: The number of nodes in the list is in the range $[0, 5 * 10^4]$.
 - Data structures: Linkedlist
- Did the problem require memory allocation? If so, describe it.
Yes, when I merge two lists, I need a sentinel node as the starting point of the linkedlist. In this case, I need to allocate memory for this sentinel node.

2. Solution Description (4 points)

- How did you solve the problem during the interview?
I used a merge sort algorithm to solve it.
 - Split the List into 2 halves: Use the slow-fast pointer technique to find the middle.
 - recursively sort each half.
 - Finally, merge the left and right halves in ascending order by comparing node values.
- What trade-offs did your solution make?
 - The solution runs in $O(n \log n)$ time, which is optimal for sorting a linked list. However, the recursion introduces overhead, especially in languages without tail call optimization.
 - Since I used top-down merge sort, the algorithm requires $O(\log n)$ space due to recursive calls. If iterative bottom-up merge sort is used, the space complexity can be $O(1)$, which could eliminate recursion overhead. In this case, we will use only constant space for storing the reference pointers.

3. C Code Submission (2 points)

Provide the C code you implemented during the interview.

I've provided solutions for 8 LeetCode questions along with a solution recording video on GitHub: https://github.com/JINGJINGPAN1/cs_5008/tree/main/mock

```
struct ListNode* get_middle(struct ListNode* head){
    struct ListNode* fast = head;
    struct ListNode* slow = head;

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

    return slow;
}

struct ListNode* merge(struct ListNode* l1, struct ListNode* l2){

}

struct ListNode* sortList(struct ListNode* head) {
    if(!head or !head->next){
        return head;
    }

    struct ListNode* mid = get_middle(head);
    struct ListNode* r = mid->next;
    struct ListNode* l = head;
    mid->next = NULL;

    l = sortList(l);
    r = sortList(r);

    return merge(l, r);
}
```

4. Complexity Analysis (2 points)

- What is the time and space complexity (Big O) of your solution?
 - Time complexity: $O(n \log n)$
 - Space complexity: $O(\log n)$
- Why does your solution have this complexity? Count operations in terms of n .
 - Time complexity: $O(n \log n)$
 - Splitting: $O(n \log n)$ ($O(n)$ per level $\times O(\log n)$ levels).
 - Merging: $O(n \log n)$ (same as splitting).
 - Overall: $O(n \log n)$.

Space complexity: $O(\log n)$.

Each recursive call to `sortList()` adds a new frame to the call stack, and the maximum depth of recursion: $O(\log n)$

- If your solution is recursive, include the recurrence relation and explain its Big O complexity.
 - (1) Time complexity: $O(n \log n)$

I've provided solutions for 8 LeetCode questions along with a solution recording video on GitHub: https://github.com/JINGJINGPAN1/cs_5008/tree/main/mock

My merge sort follows this divide-and-conquer structure: $T(n) = 2T(n/2) + O(n)$
 $2T(n/2)$: Two recursive calls on halves of size $n/2$.

$O(n)$: Work done at each level (splitting + merging).

- Level 0: $O(n)$ work (merge two halves of size $n/2$).
- Level 1: $2 \times O(n/2) = O(n)$ (merge four halves of size $n/4$).
- ...
- Level k : $2^k \times O(n/2^k) = O(n)$ (each level does $O(n)$ total work).
- Total levels: $\log_2 n$ (since we halve the list each time).

Therefore, Total time = Work per level \times Number of levels = $O(n) \times O(\log n) = O(n \log n)$.

So, Splitting: $O(n \log n)$ ($O(n)$ per level $\times O(\log n)$ levels). Merging: $O(n \log n)$ (same as splitting). Overall, time complexity is $O(n \log n)$.

(2)Space Complexity: $O(\log n)$

Each recursive call to `sortList()` uses stack space for local variables (`middle`, `l`, `r`) and the maximum stack depth: $O(\log n)$ (since the list is split in half each time).

So, space Complexity is $O(\log n)$.

5. Optimality of the Solution (2 points)

- Is there a better solution than the one you provided?

Yes! An iterative (bottom-up) merge sort improves my solution by reducing space complexity from $O(\log n)$ to $O(1)$ while keeping time complexity at $O(n \log n)$.

We can start with sublists of size 1 (each node is trivially sorted). And iteratively merge sublists in passes, doubling their size each time:

For example:

Pass 1: Merge sublists of size 1 \rightarrow size 2.

Pass 2: Merge sublists of size 2 \rightarrow size 4.

...

Until the entire list is merged.

The steps can be as the following:

- (1) Compute the list length ($O(n)$).
- (2) Merge sublists in increasing powers of 2:
 - Use a dummy node to simplify pointer management.
 - Traverse the list, merging adjacent sublists.

6. Reflection on Performance (2 points)

- What do you think you did well during the interview?

I think that I did well during this interview.

- I clearly explained the problem, my approach, and how my solution works
- I identified relevant edge cases and constraints.

I've provided solutions for 8 LeetCode questions along with a solution recording video on GitHub: https://github.com/JINGJINGPAN1/cs_5008/tree/main/mock

- I accurately describe the time and space complexity of your solution.
- I addressed the interviewers' questions thoughtfully and to the best of your ability.
- I Successfully live-code specific parts of the solution such as sort list and get_middle function while explaining my thought process.
- What (if anything) do you wish you had done differently?
 - I hope that I can have a better understanding of the interviewers' questions and not let them repeat it.
 - I hope that I have sufficient time to implement the merge function.

7. Strengths in Coding Interviews (2 points)

- What are your greatest strengths and assets in coding interviews and similar environments?
 - Clean and Modular Code: Write readable, well-structured code with meaningful variable names and helper functions. For example: Separating merge() and getMiddle() logic in merge sort.
 - Strong communication: quickly clarify requirements before solving and explain logic clearly while coding

8. Areas for Improvement (2 points)

- What aspects of your skills or performance need the most improvement to prepare for co-op or internship interviews?
 - Weakness1: Occasionally rush through problems or miss optimizations under time constraints.
 - Weakness2: Sometimes a little bit stressful during the interview process

9. Improvement Plan (2 points)

- What is your plan for practicing and improving the areas you identified for future coding interviews?
 - Plan1: Practice timed mock interviews (e.g., Pramp, Interviewing.io) or mock with peers.
 - Plan2: Daily 3 LeetCode tasks on different topics in order to get more familiar with algorithms.