0148. 排序链表

ITCharge 大约 13 分钟

- 标签：链表、双指针、分治、排序、归并排序
- 难度：中等

题目链接

- 0148. 排序链表 - 力扣

题目大意

描述：给定链表的头节点 `head` 。

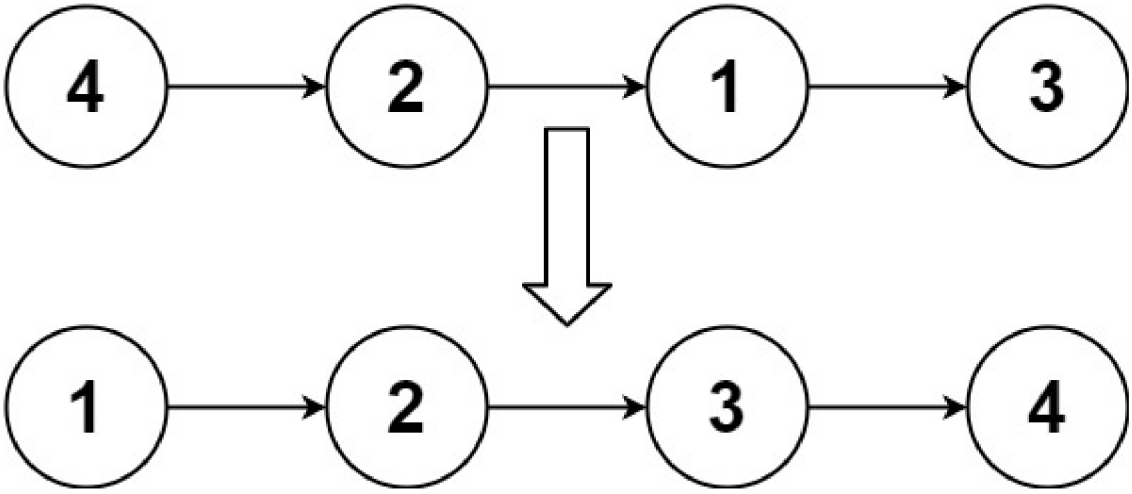
要求：按照升序排列并返回排序后的链表。

说明：

- 链表中节点的数目在范围 $[0, 5 * 10^{}$]。
- $-10^5 \leq Node.val \leq 10^5$。

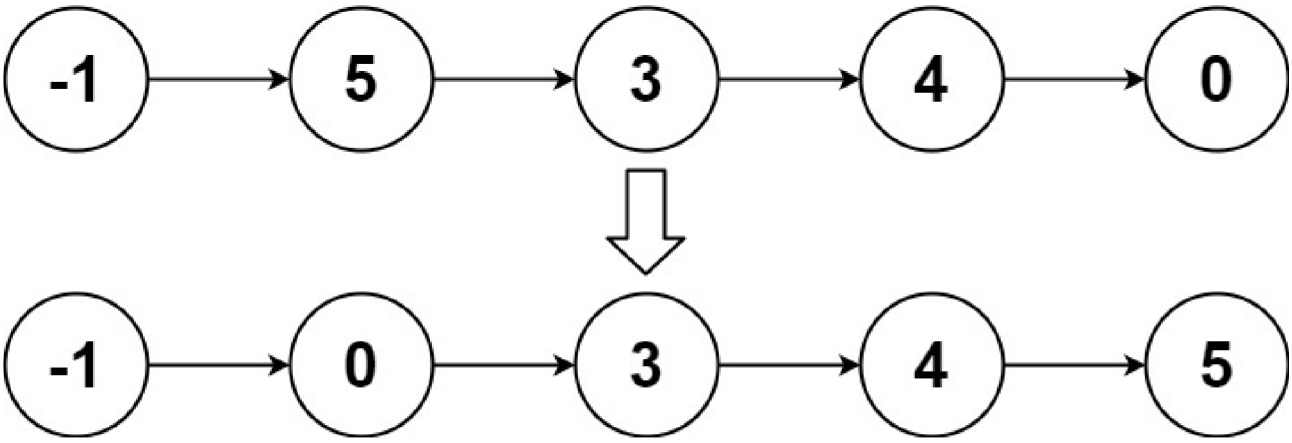
示例：

- 示例 1：



```py
输入：head = [4,2,1,3]
输出：[1,2,3,4]
```

- 示例 2：




```py
输入：head = [-1,5,3,4,0]
输出：[-1,0,3,4,5]
```

解题思路

思路 1：链表冒泡排序（超时）

1. 使用三个指针 `node_i` 、 `node_j` 和 `tail` 。其中 `node_i` 用于控制外循环次数，循环次数为链节点个数（链表长度）。 `node_j` 和 `tail` 用于控制内循环次数和循环结束位置。
2. 排序开始前，将 `node_i` 、 `node_j` 置于头节点位置。 `tail` 指向链表末尾，即 `None` 。
3. 比较链表中相邻两个元素 `node_j.val` 与 `node_j.next.val` 的值大小，如果 `node_j.val > node_j.next.val` ，则值相互交换。否则不发生交换。然后向右移动 `node_j` 指针，直到 `node_j.next == tail` 时停止。
4. 一次循环之后，将 `tail` 移动到 `node_j` 所在位置。相当于 `tail` 向左移动了一位。此时 `tail` 节点右侧为链表中最大的链节点。
5. 然后移动 `node_i` 节点，并将 `node_j` 置于头节点位置。然后重复第 3、4 步操作。
6. 直到 `node_i` 节点移动到链表末尾停止，排序结束。

7. 返回链表的头节点 `head` 。

思路 1：代码

```py
class Solution:
    def bubbleSort(self, head: ListNode):
        node_i = head
        tail = None
        # 外层循环次数为 链表节点个数
        while node_i:
            node_j = head
            while node_j and node_j.next != tail:
                if node_j.val > node_j.next.val:
                    # 交换两个节点的值
                    node_j.val, node_j.next.val = node_j.next.val, node_j.val
                node_j = node_j.next
            # 尾指针向前移动 1 位，此时尾指针右侧为排好序的链表
            tail = node_j
            node_i = node_i.next

        return head

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.bubbleSort(head)
```

思路 1：复杂度分析

- **时间复杂度**：$O(n^2)$。
- **空间复杂度**：$O(1)$。

思路 2：链表选择排序（超时）

1. 使用两个指针 `node_i` 、 `node_j` 。 `node_i` 既可以用于控制外循环次数，又可以作为当前未排序链表的第一个链节点位置。
2. 使用 `min_node` 记录当前未排序链表中值最小的链节点。
3. 每一趟排序开始时，先令 `min_node = node_i` （即暂时假设链表中 `node_i` 节点为值最小的节点，经过比较后再确定最小值节点位置）。

4. 然后依次比较未排序链表中 `node_j.val` 与 `min_node.val` 的值大小。如果 `node_j.val < min_node.val` ，则更新 `min_node` 为 `node_j` 。
5. 这一趟排序结束时，未排序链表中最小值节点为 `min_node` ，如果 `node_i != min_node` ，则将 `node_i` 与 `min_node` 值进行交换。如果 `node_i == min_node` ，则不用交换。
6. 排序结束后，继续向右移动 `node_i` ，重复上述步骤，在剩余未排序链表中寻找最小的链节点，并与 `node_i` 进行比较和交换，直到 `node_i == None` 或者 `node_i.next == None` 时，停止排序。
7. 返回链表的头节点 `head` 。

思路 2：代码

```py
class Solution:
    def sectionSort(self, head: ListNode):
        node_i = head
        # node_i 为当前未排序链表的第一个链节点
        while node_i and node_i.next:
            # min_node 为未排序链表中的值最小节点
            min_node = node_i
            node_j = node_i.next
            while node_j:
                if node_j.val < min_node.val:
                    min_node = node_j
                node_j = node_j.next
            # 交换值最小节点与未排序链表中第一个节点的值
            if node_i != min_node:
                node_i.val, min_node.val = min_node.val, node_i.val
            node_i = node_i.next

        return head

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.sectionSort(head)
```

思路 2：复杂度分析

- **时间复杂度**：$O(n^2)$。
- **空间复杂度**：$O(1)$。

思路 3：链表插入排序（超时）

1. 先使用哑节点 `dummy_head` 构造一个指向 `head` 的指针，使得可以从 `head` 开始遍历。
2. 维护 `sorted_list` 为链表的已排序部分的最后一个节点，初始时， `sorted_list = head` 。
3. 维护 `prev` 为插入元素位置的前一个节点，维护 `cur` 为待插入元素。初始时， `prev = head` ， `cur = head.next` 。
4. 比较 `sorted_list` 和 `cur` 的节点值。
   - 如果 `sorted_list.val <= cur.val` ，说明 `cur` 应该插入到 `sorted_list` 之后，则将 `sorted_list` 后移一位。
   - 如果 `sorted_list.val > cur.val` ，说明 `cur` 应该插入到 `head` 与 `sorted_list` 之间。则使用 `prev` 从 `head` 开始遍历，直到找到插入 `cur` 的位置的前一个节点位置。然后将 `cur` 插入。
5. 令 `cur = sorted_list.next` ，此时 `cur` 为下一个待插入元素。
6. 重复 4、5 步骤，直到 `cur` 遍历结束为空。返回 `dummy_head` 的下一个节点。

思路 3：代码

```py
class Solution:
    def insertionSort(self, head: ListNode):
        if not head or not head.next:
            return head

        dummy_head = ListNode(-1)
        dummy_head.next = head
        sorted_list = head
        cur = head.next

        while cur:
            if sorted_list.val <= cur.val:
                # 将 cur 插入到 sorted_list 之后
                sorted_list = sorted_list.next
            else:
```

```
                prev = dummy_head
                while prev.next.val <= cur.val:
                    prev = prev.next
                # 将 cur 到链表中间
                sorted_list.next = cur.next
                cur.next = prev.next
                prev.next = cur
            cur = sorted_list.next

        return dummy_head.next

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.insertionSort(head)
```

思路 3：复杂度分析

- **时间复杂度**：$O(n^2)$。
- **空间复杂度**：$O(1)$。

思路 4：链表归并排序（通过）

1. **分割环节**：找到链表中心链节点，，、，心节点将链表断开，并递归进行分割。
    1. 使用快慢指针 `fast = head.next` 、 `slow = head` ，让 `fast` 每次移动 `2` 步， `slow` 移动 `1` 步，移动到链表末尾，从而找到链表中心链节点，即 `slow` 。
    2. 从中心位置将链表从中心位置分为左右两个链表 `left_head` 和 `right_head` ，并从中心位置将其断开，即 `slow.next = None` 。
    3. 对左右两个链表分别进行递归分割，直到每个链表中只包含一个链节点。
2. **归并环节**：将递归后的链表进行两两归并，完成一遍后每个子链表长度加倍。重复进行归并操作，直到得到完整的链表。
    1. 使用哑节点 `dummy_head` 构造一个头节点，并使用 `cur` 指向 `dummy_head` 用于遍历。
    2. 比较两个链表头节点 `left` 和 `right` 的值大小。将较小的头节点加入到合并后的链表中。并向后移动该链表的头节点指针。
    3. 然后重复上一步操作，直到两个链表中出现链表为空的情况。
    4. 将剩余链表插入到合并中的链表中。
    5. 将哑节点 `dummy_dead` 的下一个链节点 `dummy_head.next` 作为合并后的头节点返回。

思路 4：代码

```py
class Solution:
    def merge(self, left, right):
        # 归并环节
        dummy_head = ListNode(-1)
        cur = dummy_head
        while left and right:
            if left.val <= right.val:
                cur.next = left
                left = left.next
            else:
                cur.next = right
                right = right.next
            cur = cur.next

        if left:
            cur.next = left
        elif right:
            cur.next = right

        return dummy_head.next

    def mergeSort(self, head: ListNode):
        # 分割环节
        if not head or not head.next:
            return head

        # 快慢指针找到中心链节点
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # 断开左右链节点
        left_head, right_head = head, slow.next
        slow.next = None
```

```
        # 归并操作
        return self.merge(self.mergeSort(left_head), self.mergeSort(right_head))

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.mergeSort(head)
```

思路 4：复杂度分析

- **时间复杂度**：$O(n \times \log_2 n)$。
- **空间复杂度**：$O(1)$。

思路 5：链表快速排序（超时）

1. 从链表中找到一个基准值 `pivot`，这里以头节点为基准值。
2. 然后通过快慢指针 `node_i`、`node_j` 在链表中移动，使得 `node_i` 之前的节点值都小于基准值，`node_i` 之后的节点值都大于基准值。从而把数组拆分为左右两个部分。
3. 再对左右两个部分分别重复第二步，直到各个部分只有一个节点，则排序结束。

> 注意：
> 虽然链表快速排序算法的平均时间复杂度为 $O(n \times \log_2 n)$。但链表快速排序算法中基准值 `pivot` 的取值做不到数组快速排序算法中的随机选择。一旦给定序列是有序链表，时间复杂度就会退化到 $O(n^2)$。这也是这道题目使用链表快速排序容易超时的原因。

思路 5：代码

py

```
class Solution:
    def partition(self, left: ListNode, right: ListNode):
        # 左闭右开，区间没有元素或者只有一个元素，直接返回第一个节点
        if left == right or left.next == right:
            return left
        # 选择头节点为基准节点
        pivot = left.val
        # 使用 node_i, node_j 双指针，保证 node_i 之前的节点值都小于基准节点值，
node_i 与 node_j 之间的节点值都大于等于基准节点值
        node_i, node_j = left, left.next

        while node_j != right:
```

```
            # 发现一个小与基准值的元素
            if node_j.val < pivot:
                # 因为 node_i 之前节点都小于基准值，所以先将 node_i 向右移动一位（此时 node_i 节点值大于等于基准节点值）
                node_i = node_i.next
                # 将小于基准值的元素 node_j 与当前 node_i 换位，换位后可以保证 node_i 之前的节点都小于基准节点值
                node_i.val, node_j.val = node_j.val, node_i.val
            node_j = node_j.next
        # 将基准节点放到正确位置上
        node_i.val, left.val = left.val, node_i.val
        return node_i

    def quickSort(self, left: ListNode, right: ListNode):
        if left == right or left.next == right:
            return left
        pi = self.partition(left, right)
        self.quickSort(left, pi)
        self.quickSort(pi.next, right)
        return left

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next:
            return head
        return self.quickSort(head, None)
```

思路 5：复杂度分析

- **时间复杂度**：$O(n \times \log_2 n)$。
- **空间复杂度**：$O(1)$。

思路 6：链表计数排序（通过）

1. 使用 `cur` 指针遍历一遍链表。找出链表中最大值 `list_max` 和最小值 `list_min` 。
2. 使用数组 `counts` 存储节点出现次数。
3. 再次使用 `cur` 指针遍历一遍链表。将链表中每个值为 `cur.val` 的节点出现次数，存入数组对应第 `cur.val - list_min` 项中。
4. 反向填充目标链表：
    1. 建立一个哑节点 `dummy_head` ，作为链表的头节点。使用 `cur` 指针指向 `dummy_head` 。

2. 从小到大遍历一遍数组 `counts` 。对于每个 `counts[i] != 0` 的元素建立一个链节点，值为 `i + list_min` ，将其插入到 `cur.next` 上。并向右移动 `cur` 。同时 `counts[i] -= 1` 。直到 `counts[i] == 0` 后继续向后遍历数组 `counts` 。

5. 将哑节点 `dummy_dead` 的下一个链节点 `dummy_head.next` 作为新链表的头节点返回。

思路 6：代码

py

```py
class Solution:
    def countingSort(self, head: ListNode):
        if not head:
            return head

        # 找出链表中最大值 list_max 和最小值 list_min
        list_min, list_max = float('inf'), float('-inf')
        cur = head
        while cur:
            if cur.val < list_min:
                list_min = cur.val
            if cur.val > list_max:
                list_max = cur.val
            cur = cur.next

        size = list_max - list_min + 1
        counts = [0 for _ in range(size)]

        cur = head
        while cur:
            counts[cur.val - list_min] += 1
            cur = cur.next

        dummy_head = ListNode(-1)
        cur = dummy_head
        for i in range(size):
            while counts[i]:
                cur.next = ListNode(i + list_min)
                counts[i] -= 1
                cur = cur.next
        return dummy_head.next

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.countingSort(head)
```

思路 6：复杂度分析

- **时间复杂度**：$O(n + k)$，其中 k 代表待排序链表中所有元素的值域。
- **空间复杂度**：$O(k)$。

思路 7：链表桶排序（通过）

1. 使用 `cur` 指针遍历一遍链表。找出链表中最大值 `list_max` 和最小值 `list_min` 。
2. 通过 `（最大值 - 最小值）/ 每个桶的大小` 计算出桶的个数，即 `bucket_count = (list_max - list_min) // bucket_size + 1` 个桶。
3. 定义数组 `buckets` 为桶，桶的个数为 `bucket_count` 个。
4. 使用 `cur` 指针再次遍历一遍链表，将每个元素装入对应的桶中。
5. 对每个桶内的元素单独排序，可以使用链表插入排序（超时）、链表归并排序（通过）、链表快速排序（超时）等算法。
6. 最后按照顺序将桶内的元素拼成新的链表，并返回。

思路 7：代码

```py
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    # 将链表节点值 val 添加到对应桶 buckets[index] 中
    def insertion(self, buckets, index, val):
        if not buckets[index]:
            buckets[index] = ListNode(val)
            return

        node = ListNode(val)
        node.next = buckets[index]
        buckets[index] = node

    # 归并环节
    def merge(self, left, right):
        dummy_head = ListNode(-1)
```

```
        cur = dummy_head
        while left and right:
            if left.val <= right.val:
                cur.next = left
                left = left.next
            else:
                cur.next = right
                right = right.next
            cur = cur.next

        if left:
            cur.next = left
        elif right:
            cur.next = right

        return dummy_head.next

    def mergeSort(self, head: ListNode):
        # 分割环节
        if not head or not head.next:
            return head

        # 快慢指针找到中心链节点
        slow, fast = head, head.
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # 断开左右链节点
        left_head, right_head = head, slow.next
        slow.next = None

        # 归并操作
        return self.merge(self.mergeSort(left_head), self.mergeSort(right_head))

    def bucketSort(self, head: ListNode, bucket_size=5):
        if not head:
            return head

        # 找出链表中最大值 list_max 和最小值 list_min
        list_min, list_max = float('inf'), float('-inf')
        cur = head
        while cur:
            if cur.val < list_min:
```

```
                list_min = cur.val
            if cur.val > list_max:
                list_max = cur.val
            cur = cur.next

        # 计算桶的个数，并定义桶
        bucket_count = (list_max - list_min) // bucket_size + 1
        buckets = [[] for _ in range(bucket_count)]

        # 将链表节点值依次添加到对应桶中
        cur = head
        while cur:
            index = (cur.val - list_min) // bucket_size
            self.insertion(buckets, index, cur.val)
            cur = cur.next

        dummy_head = ListNode(-1)
        cur = dummy_head
        # 将元素依次出桶，并拼接成有序链表
        for bucket_head in buckets:
            bucket_cur = self.mergeSort(bucket_head)
            while bucket_cur:
                cur.next = bucket_cur
                cur = cur.next
                bucket_cur = bucket_cur.next

        return dummy_head.next

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.bucketSort(head)
```

思路 7：复杂度分析

- **时间复杂度**：$O(n)$。
- **空间复杂度**：$O(n+m)$。m 为桶的个数。

思路 8：链表基数排序（解答错误，普通链表基数排序只适合非负数）

1. 使用 `cur` 指针遍历链表，获取节点值位数最长的位数 `size` 。
2. 从个位到高位遍历位数。因为 `0 ~ 9` 共有 `10` 位数字，所以建立 `10` 个桶。

3. 以每个节点对应位数上的数字为索引，将节点值放入到对应桶中。
4. 建立一个哑节点 `dummy_head` ，作为链表的头节点。使用 `cur` 指针指向 `dummy_head` 。
5. 将桶中元素依次取出，并根据元素值建立链表节点，并插入到新的链表后面。从而生成新的链表。
6. 之后依次以十位，百位，…，直到最大值元素的最高位处值为索引，放入到对应桶中，并生成新的链表，最终完成排序。
7. 将哑节点 `dummy_dead` 的下一个链节点 `dummy_head.next` 作为新链表的头节点返回。

思路 8：代码

```py
class Solution:
    def radixSort(self, head: ListNode):
        # 计算位数最长的位数
        size = 0
        cur = head
        while cur:
            val_len = len(str(cur.val))
            if val_len > size:
                size = val_len
            cur = cur.next

        # 从个位到高位遍历位数
        for i in range(size):
            buckets = [[] for _ in range(10)]
            cur = head
            while cur:
                # 以每个节点对应位数上的数字为索引，将节点值放入到对应桶中
                buckets[cur.val // (10 ** i) % 10].append(cur.val)
                cur = cur.next

            # 生成新的链表
            dummy_head = ListNode(-1)
            cur = dummy_head
            for bucket in buckets:
                for num in bucket:
                    cur.next = ListNode(num)
                    cur = cur.next
            head = dummy_head.next
```

```
        return head

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.radixSort(head)
```

思路 8：复杂度分析

- **时间复杂度**：$O(n \times k)$。其中 n 是待排序元素的个数，k 是数字位数。k 的大小取决于数字位的选择（十进制位、二进制位）和待排序元素所属数据类型全集的大小。
- **空间复杂度**：$O(n + k)$。

参考资料

- 【文章】单链表的冒泡排序_zhao_miao的博客 - CSDN博客
- 【文章】链表排序总结（全）（C++）- 阿祭儿 - CSDN博客
- 【题解】快排、冒泡、选择排序实现列表排序 - 排序链表 - 力扣
- 【题解】归并排序+快速排序 - 排序链表 - 力扣
- 【题解】排序链表（递归+迭代）详解 - 排序链表 - 力扣
- 【题解】Sort List （归并排序链表`　排序链表 - 力扣