# 题目

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

**示例 1:**

输入: 4->2->1->3

输出: 1->2->3->4

**示例 2:**

输入: -1->5->3->4->0

输出: -1->0->3->4->5

# 分析

## 方法一：归并排序

class Solution {

public:

ListNode \* sortList(ListNode \* head)

{

return (head == NULL)? NULL: mergeSort(head);

}

private:

ListNode \* findMid(ListNode \* head)

{

ListNode \* slow = head;

ListNode \* fast = head;

ListNode \* previous = NULL;

while (fast != NULL && fast->next != NULL)

{

previous = slow;

slow = slow->next;

fast = fast->next->next;

}

// split the list into two parts

previous->next = NULL;

return slow;

}

ListNode \* mergeTwoLists(ListNode \* l1, ListNode \* l2)

{

if(l1 == NULL) return l2;

if(l2 == NULL) return l1;

if(l1->val < l2->val){

l1->next = mergeTwoLists(l1->next,l2);

return l1;

}else{

l2->next = mergeTwoLists(l1,l2->next);

return l2;

}

}

ListNode \* mergeSort(ListNode \* head)

{

if (head->next == NULL) return head;

ListNode \* mid = findMid(head);

ListNode \* l1 = mergeSort(head);

ListNode \* l2 = mergeSort(mid);

return mergeTwoLists(l1, l2);

}

};

## 方法二：自顶向下归并排序

对链表自顶向下归并排序的过程如下。

1、找到链表的中点，以中点为分界，将链表拆分成两个子链表。寻找链表的中点可以使用快慢指针的做法，快指针每次移动2步，慢指针每次移动1步，当快指针到达链表末尾时，慢指针指向的链表节点即为链表的中点。

2、对两个子链表分别排序。

3、将两个排序后的子链表合并，得到完整的排序后的链表。可以使用「21. 合并两个有序链表」的做法，将两个有序的子链表进行合并。

上述过程可以通过递归实现。递归的终止条件是链表的节点个数小于或等于1，即当链表为空或者链表只包含1个节点时，不需要对链表进行拆分和排序。

**代码：**

class Solution {

public:

ListNode\* sortList(ListNode\* head) {

return sortList(head, nullptr);

}

ListNode\* sortList(ListNode\* head, ListNode\* tail) {

if (head == nullptr) {

return head;

}

if (head->next == tail) {

head->next = nullptr;

return head;

}

ListNode\* slow = head, \*fast = head;

while (fast != tail) {

slow = slow->next;

fast = fast->next;

if (fast != tail) {

fast = fast->next;

}

}

ListNode\* mid = slow;

return merge(sortList(head, mid), sortList(mid, tail));

}

ListNode\* merge(ListNode\* head1, ListNode\* head2) {

ListNode\* dummyHead = new ListNode(0);

ListNode\* temp = dummyHead, \*temp1 = head1, \*temp2 = head2;

while (temp1 != nullptr && temp2 != nullptr) {

if (temp1->val <= temp2->val) {

temp->next = temp1;

temp1 = temp1->next;

} else {

temp->next = temp2;

temp2 = temp2->next;

}

temp = temp->next;

}

if (temp1 != nullptr) {

temp->next = temp1;

} else if (temp2 != nullptr) {

temp->next = temp2;

}

return dummyHead->next;

}

};

**复杂度分析：**

时间复杂度：O(nlogn)，其中n是链表的长度。

空间复杂度：O(logn)，其中n是链表的长度。空间复杂度主要取决于递归调用的栈空间。

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode() : val(0), next(nullptr) {}

 \*     ListNode(int x) : val(x), next(nullptr) {}

 \*     ListNode(int x, ListNode \*next) : val(x), next(next) {}

 \* };

 \*/

class Solution {

public:

    ListNode\* sortList(ListNode\* head) {

        return sort(head,nullptr);

    }

    ListNode \*sort(ListNode \*head,ListNode \*tail)

    {

        if(nullptr==head)

            return head;

        if(head->next==tail)

        {

            head->next = nullptr;

            return head;

        }

        ListNode \*slow=head,\*fast=head;

        while(fast!=tail)

        {

            slow = slow->next;

            fast = fast->next;

            if(fast!=tail)

                fast = fast->next;

        }

        ListNode \*mid = slow;

        return merge(sort(head,mid),sort(mid,tail));

    }

    ListNode \*merge(ListNode \*head1,ListNode \*head2)

    {

        ListNode \*dummyNode = new ListNode(0);

        ListNode \*tmpNode = dummyNode;

        ListNode \*p=head1,\*q=head2;

        while(nullptr!=p && nullptr!=q)

        {

            if(p->val <= q->val)

            {

                tmpNode->next = p;

                p = p->next;

            }

            else

            {

                tmpNode->next = q;

                q = q->next;

            }

            tmpNode = tmpNode->next;

        }

        if(nullptr!=p)

        {

            tmpNode->next = p;

        }

        else if(nullptr!=q)

        {

            tmpNode->next = q;

        }

        return dummyNode->next;

    }

};