## Merge Two Sorted Lists

```
class Solution {
    public 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(l2.next, l1);
            return l2;
        }
    }
}
```

## Reverse Nodes in k-Group

子问题和原问题的结构完全相同，
这就是所谓的递归性质

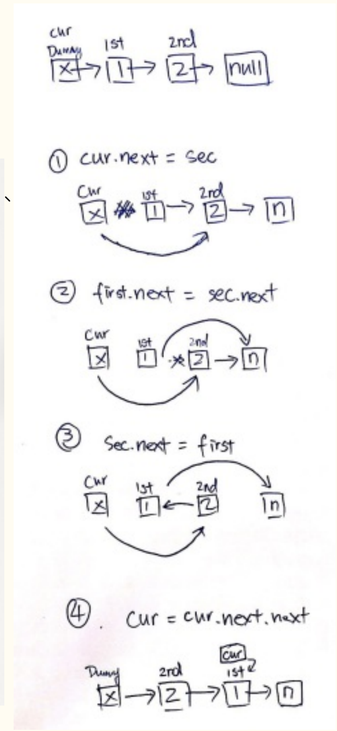
dummy head;
return dummy.next;
d -> 1(tail) -> 2(curr) -> 3(next) -> 4(last)
reverse 1, 2, 3, return tail(1);

https://www.youtube.com/watch?v=pLx1VP-FnuY

## Swap Nodes In Pairs

```
//  1 -> 2 -> 3 -> 4
//  2 -> 1 -> 4 -> 3
//  (2 tail) -> (1 head) -> swapPairs(3 -> 4)
//  return tail
class Solution {
    public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode tail = head.next;
        head.next = swapPairs(head.next.next);
        tail.next = head;
        return tail;
    }
}
```

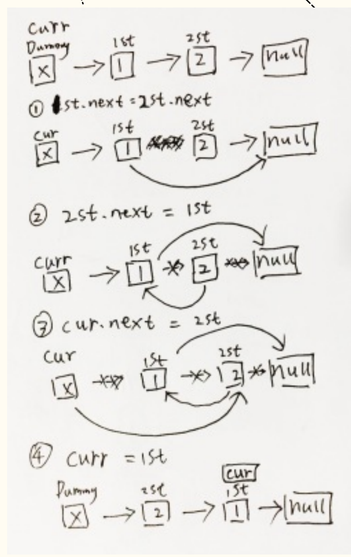
核心是 反复交换 prev 到 last node(k + 1) 之间的节点，并每次更新 prev, 直到 prev 为 null
两个重要指针
dummy.next 指向 curr
tail.next 指向 next




## 反转列表

```
// 1 -> 2 -> null
// null <- 1 <- 2
// 声明一个 prev 变量, 从 prev = null 开始
// 声明一个 curr 变量, 从 curr = head 开始
// 存一下 curr.next, 将 curr 指向前一个变量, 将 prev = curr
// 将 curr.next 作为下一个 curr
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

```
//  递归查找并返回原 list's tail
//  返回之前将 tail 的 next 改为 head -> head.next.next = head;
//  head.next = null;
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode tail = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return tail;
    }
}
```




## Linked List Cycle

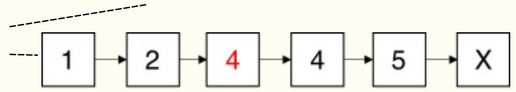
获取闭合点
1.先检查是否闭环,
或不重逢 return null

2.fast 为重逢点，将slow重置为 head
3.while 将快慢指针统一每次走一步
4.重合点就是闭合点

判断是否有闭环
出发点都是 head 的快慢指针
slow 一次一步,
fast 一次两步
如果 slow 与 fast 不相遇 则一直走
如果 fast or fast.next == null 说明没有环:
1 -> 2 -> 3 -> null (fast.next 为 null)
1 -> 2 -> 3 -> 4 -> null (fast 为 null)
p.s 带下划线的是 fast 的移动轨迹

## 链表操作

- **Middle of the Linked List**
  - 快慢指针
    - 慢指针在中间时，快指针已到终点
    - 注意处理
    - **odd nodes: let right half smaller**
  - 遍历链表得到总数，算出中间点，遍历到中间点
- **Delete Node in a Linked List**
  - 将当前节点的 val 以及 next 改为后一个节点




- **Remove Duplicates from Sorted List**
  - curr 与 curr.next 对比
  - 如果重复 curr.next = curr.next.next;
  - 不重复则 curr = curr.next;
- **Remove Linked List Elements**
  - 需要改变当前被删节点的前后节点
  - 所以创建 dummy 头, 返回 dummy.next
- **Intersection of Two Linked Lists**
  - 让两个链表从同距离末尾同等距离的位置开始遍历。这个位置只能是较短链表的头结点位置。
  - 为此，我们必须消除两个链表的长度差