1.

输入：顺序表 A，长度为 n
输出：删除重复元素后的顺序表 A

1. 初始化一个空的哈希表 visited
2. 初始化一个索引 i = 0
3. 遍历顺序表 A：
   4. 如果 A[i] 不在 visited 中：
      5. 将 A[i] 加入到 visited 中
      6. i 增加 1
   7. 否则：
      8. 从顺序表 A 中删除 A[i]（即跳过该元素，而不增加 i）

9. 返回修改后的顺序表 A

时间复杂度分析:
遍历顺序表的时间复杂度为 $O(n)$, 因为我们需要遍历每个元素。
每次查找和插入哈希表的平均时间复杂度为 $O(1)$, 最坏时间复杂度是 $O(n)$.
从顺序表中删除$A[i]$的时间复杂度是 $O(n)$.
所以，总的时间复杂度为 $O(n^2)$.
空间复杂度分析:
由于我们需要使用一个哈希表来存储已经访问过的元素，哈希表的大小最多为 $O(n)$.
因此，空间复杂度为 $O(n)$.

2.

```
function insertAfter(node, newNode):
    // 1. 将新节点的 prev 指针指向当前节点
    newNode.prev = node

    // 2. 将新节点的 next 指针指向当前节点的 next 节点
    newNode.next = node.next

    // 3. 如果当前节点的 next 节点不为空，将它的 prev 指针指向新节点
    if node.next != null:
        node.next.prev = newNode

    // 4. 将当前节点的 next 指针指向新节点
    node.next = newNode
```

3.

```
function detectCycle(head):
    if head == null or head.next == null:
        return null  // 空链表或单个节点链表，没有环

    // 初始化快慢指针
    slow = head
    fast = head

    // 判断是否有环
    while fast != null and fast.next != null:
        slow = slow.next      // 慢指针每次走一步
        fast = fast.next.next // 快指针每次走两步

        if slow == fast: // 快慢指针相遇，存在环
            // 重新将 slow 指针指向 head
            slow = head

            // 两个指针每次走一步，直到相遇
            while slow != fast:
                slow = slow.next
                fast = fast.next

            // 此时 slow 和 fast 相遇的地方就是环的入口
            return slow

    // 没有环
    return null
```