以下都使用接近 C++ 代码的伪代码.

1

其伪代码如下:

```
void my_unique(List lst){
    for (int i = 1; i < lst.size(); ++i){
        while (lst[i] == lst[i-1] && i < lst.size())
            lst.remove(i);
    }
}
```

虽然代码中有两重循环，但内层循环调用 `remove` 函数时会导致 `size()` 减一，因此最内层的 `remove` 至多执行 $n-1$ 次. 另一方面，顺序表的删除是 $O(n)$ 的，因此这份代码的时间复杂度为 $O(n^2)$.

由于顺序表的实现细节被隐藏，我们删除元素时只能调用 `remove` 函数，最坏情况下又不得不调用 $n-1$ 次，因此 $O(n^2)$ 已经是时间渐进最优的解法，尽管显而易见地有大量冗余操作.

但是，如果将 `my_unique` 作为成员函数实现，直接操纵内部细节，则不难给出一个更好的解法:

```
void List::my_unqiue(){
    int i = 0, j = 1;
    while (true){
        while (aList[i] == aList[j] && j < size()) ++j;
        if (j < size()) aList[++i] = aList[j++];
        else break;
    }
    m_size = i+1;
}
```

由于 `j` 至多自增 $n-1$ 次，所有操作都是 (1) 的，因此这份代码是 $O(n)$ 的.

2

由于链表本质上是一个递归的数据结构，我首先考虑使用递归调用来完成这个问题:

```
Node* reverse_helper(Node pre, Node cur);

Node* reverse(Node* head){
    Node* t = head->next;
    head->next = nullptr;
    return reverse_helper(head, t);
}

Node* reverse_helper(Node* pre, Node* cur){
    if (cur == nullptr) return pre;
    Node* nxt = cur->next;
    cur->next = pre;
    return reverse_helper(cur, nxt);
}
```

为了使函数参数更方便递归，我使用了辅助函数.

递归算法的结构清晰易懂，但递归本身会有 $O(n)$ 的开销，因此我又将其改写为了等价的循环实现：

```
Node reverse(Node head){
    Node* pre = head;
    Node* cur = head->next;
    Node* nxt;
    while (cur != nullptr){
        nxt = cur->next;
        cur->next = pre;
        pre = cur, cur = nxt;
    }
    head->next = nullptr;
    return pre;
}
```

由于链表只会被遍历一遍，因此其时间复杂度为 $O(n)$. 由于全程只使用了 `pre` `cur` `nxt` 这三个变量，因此其空间复杂度为 $O(1)$.

3

其伪代码如下：

```
bool hasLoop(List L){
    if (L.head == nullptr) return false;    // head is a pointer to the first
node, not the first node itself
    Node* slow = L.head;
    Node* fast = L.head;
    while (fast != nullptr){
        slow = slow->next;
        if (fast != nullptr) fast = fast->next;
        if (fast != nullptr) fast = fast->next;
        if (fast == slow) return true;
    }
    return false;
}
```

这个算法的执行过程是：

1. 如果链表为空，显然无环，算法结束.
2. 如果链表非空，创建两个指针 `slow` 和 `fast`，指向链表的第一个节点.
3. 不断循环令 `slow` 步进一个单位，`fast` 步进两个单位，并检查这样两件事：
    1. `fast` 是否到达了链表尾. 如果到达，说明无环，算法结束.
    2. `slow` 和 `fast` 是否指向同一节点. 如果是，由于两者步长不同，必然有环，算法结束.

当有环时，由于步长差 1，进入环后 `fast` 每经历一轮循环会“接近” `slow` 一个单位，因此显然经历 $O(n)$ 轮循环后必有 `slow == fast`.

综上，这个算法的时间复杂度为 $O(n)$.

另外，显然其空间复杂度为 $O(1)$.