

WEEK1

Contents

- 1. Contains Duplicate - Easy**
- 2. Best Time to Buy and Sell Stock - Easy**
- 3. Range Sum Query - Immutable - Easy**
- 4. Linked List Cycle - Easy**
- 5. Middle of the Linked List - Easy**
- 6. Reverse Linked List - Easy**
- 7. Remove Linked List Elements - Easy**
- 8. Remove Duplicates from Sorted List - Easy**
- 9. Merge Two Sorted Lists - Easy**

Contains Duplicate

Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

Example 1:

Input: `nums = [1,2,3,1]`

Output: `true`

Explanation:

The element 1 occurs at the indices 0 and 3.

Example 2:

Input: `nums = [1,2,3,4]`

Output: `false`

Explanation:

All elements are distinct.

Example 3:

Input: `nums = [1,1,1,3,3,4,3,2,4,2]`

Output: `true`

Constraints:

- `1 <= nums.length <= 105`
- `-109 <= nums[i] <= 109`

```
class Solution { public: bool containsDuplicate(vector& nums) { } };
```

Best Time to Buy and Sell Stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

Input: `prices = [7,1,5,3,6,4]` **Output:** 5 **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]` **Output:** 0 **Explanation:** In this case, no transactions are done and the max profit = 0.

Constraints:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^4$

```
class Solution { public: int maxProfit(vector& prices) { } };
```

Range Sum Query - Immutable

Given an integer array `nums`, handle multiple queries of the following type:

1. Calculate the **sum** of the elements of `nums` between indices `left` and `right` **inclusive** where `left <= right`.

Implement the `NumArray` class:

- `NumArray(int[] nums)` Initializes the object with the integer array `nums`.
- `int sumRange(int left, int right)` Returns the **sum** of the elements of `nums` between indices `left` and `right` **inclusive** (i.e. `nums[left] + nums[left + 1] + ... + nums[right]`).

Example 1:

Input ["NumArray", "sumRange", "sumRange", "sumRange"] [[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]

Output [null, 1, -1, -3] **Explanation** `NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);`

`numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1` `numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1` `numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3`

Constraints:

- `1 <= nums.length <= 104`
- `-105 <= nums[i] <= 105`
- `0 <= left <= right < nums.length`
- At most `104` calls will be made to `sumRange`.

`class NumArray { public: NumArray(vector& nums) { } int sumRange(int left, int right) { } };` **** Your NumArray object will be instantiated and called as such: `* NumArray* obj = new NumArray(nums); * int param_1 = obj->sumRange(left,right); */`**

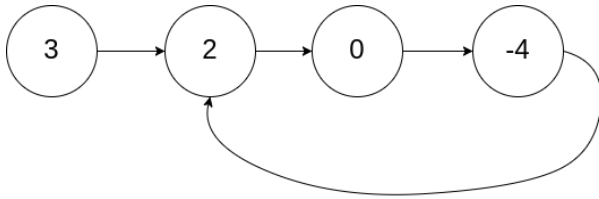
Linked List Cycle

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.**

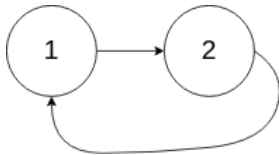
Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

Example 1:



Input: `head = [3,2,0,-4]`, `pos = 1` **Output:** `true` **Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: `head = [1,2]`, `pos = 0` **Output:** `true` **Explanation:** There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:



Input: `head = [1]`, `pos = -1` **Output:** `false` **Explanation:** There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range $[0, 10^4]$.
- $-10^5 \leq \text{Node.val} \leq 10^5$
- `pos` is `-1` or a **valid index** in the linked-list.

Follow up: Can you solve it using $O(1)$ (i.e. constant) memory?

```
/** Definition for singly-linked list. * struct ListNode { * int val; * ListNode *next; * ListNode(int x) : val(x), next(NULL) {} * }; */ class Solution { public: bool hasCycle(ListNode *head) { } };
```

Middle of the Linked List

Given the head of a singly linked list, return *the middle node of the linked list*.

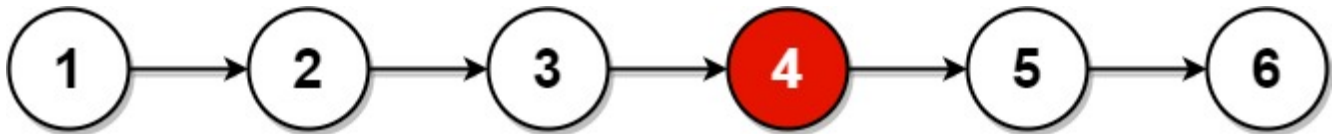
If there are two middle nodes, return **the second middle node**.

Example 1:



Input: head = [1,2,3,4,5] **Output:** [3,4,5] **Explanation:** The middle node of the list is node 3.

Example 2:



Input: head = [1,2,3,4,5,6] **Output:** [4,5,6] **Explanation:** Since the list has two middle nodes with values 3 and 4, we return the second one.

Constraints:

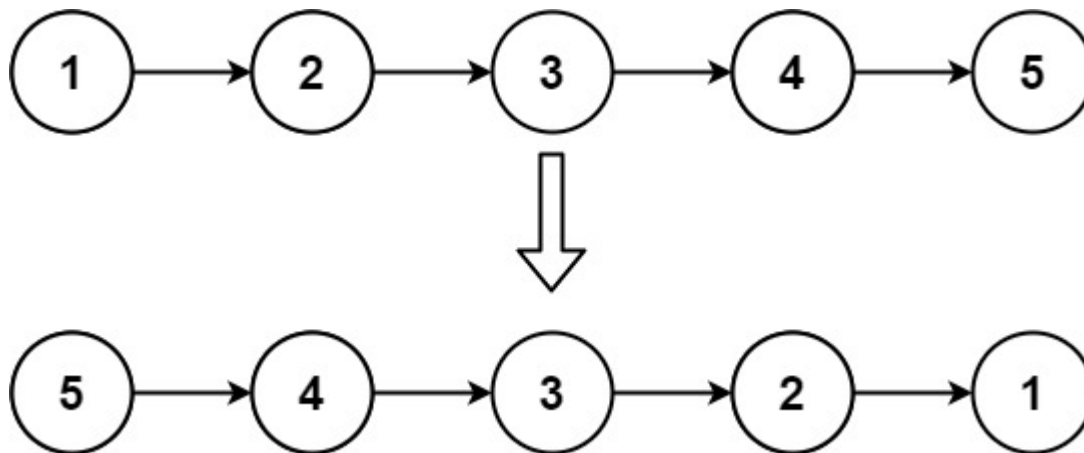
- The number of nodes in the list is in the range [1, 100].
- 1 ≤ Node.val ≤ 100

```
/** * 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* middleNode(ListNode* head) { }}
```

Reverse Linked List

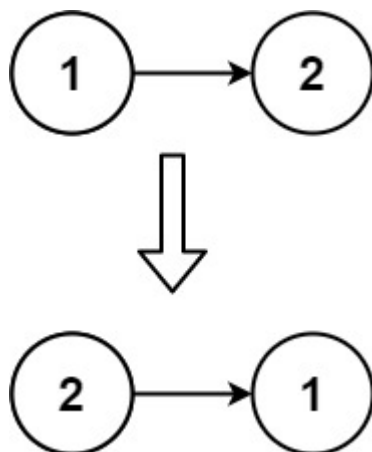
Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:



Input: head = [1,2,3,4,5] **Output:** [5,4,3,2,1]

Example 2:



Input: head = [1,2] **Output:** [2,1]

Example 3:

Input: head = [] **Output:** []

Constraints:

- The number of nodes in the list is the range [0, 5000].
- $-5000 \leq \text{Node.val} \leq 5000$

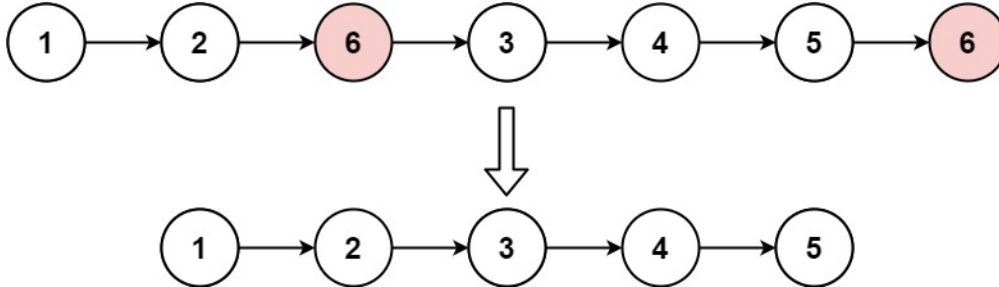
Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?


```
/** * 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* reverseList(ListNode* head) { } };
```

Remove Linked List Elements

Given the head of a linked list and an integer `val`, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

Example 1:



Input: head = [1,2,6,3,4,5,6], val = 6 **Output:** [1,2,3,4,5]

Example 2:

Input: head = [], val = 1 **Output:** []

Example 3:

Input: head = [7,7,7,7], val = 7 **Output:** []

Constraints:

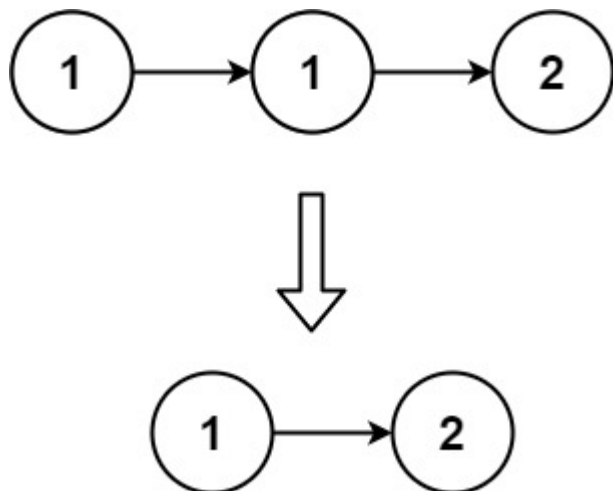
- The number of nodes in the list is in the range $[0, 10^4]$.
- $1 \leq \text{Node.val} \leq 50$
- $0 \leq \text{val} \leq 50$

```
/** * 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* removeElements(ListNode* head, int val) {} };
```

Remove Duplicates from Sorted List

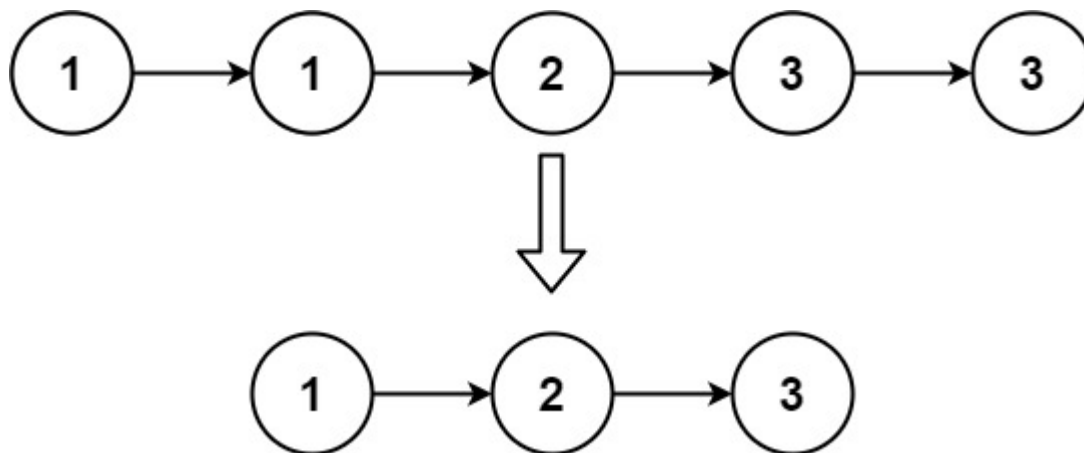
Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list **sorted** as well.

Example 1:



Input: head = [1,1,2] Output: [1,2]

Example 2:



Input: head = [1,1,2,3,3] Output: [1,2,3]

Constraints:

- The number of nodes in the list is in the range $[0, 300]$.
- $-100 \leq \text{Node.val} \leq 100$
- The list is guaranteed to be **sorted** in ascending order.

```
/** * 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* deleteDuplicates(ListNode* head) { }
```

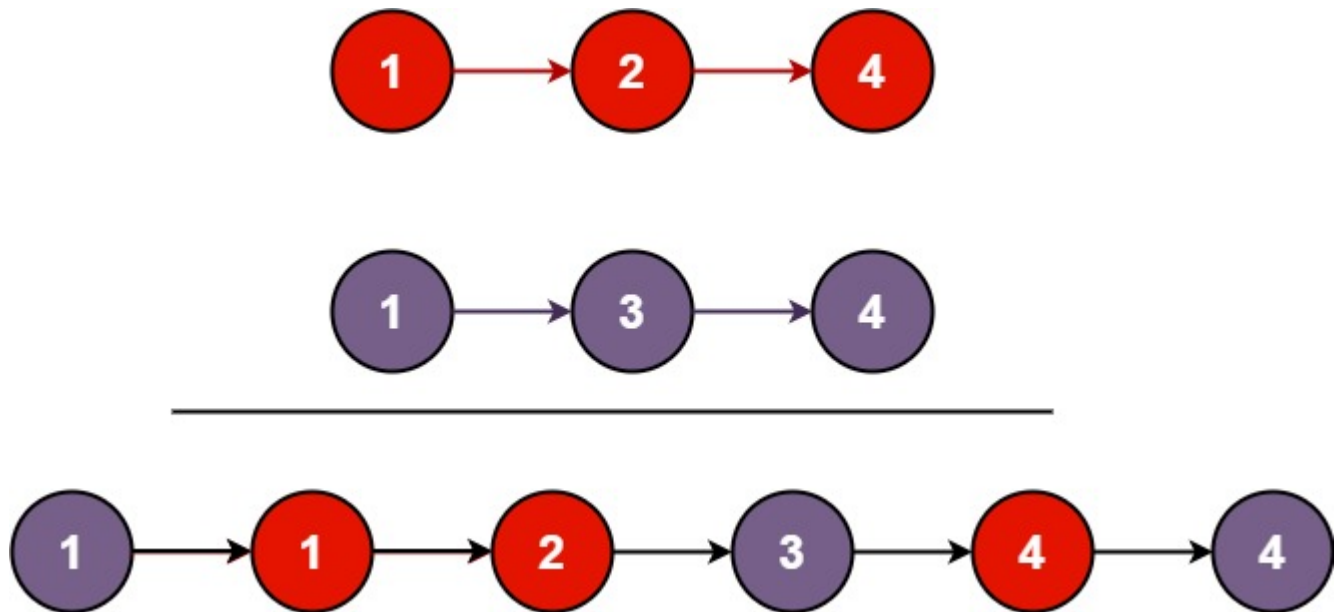
Merge Two Sorted Lists

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



Input: `list1 = [1,2,4]`, `list2 = [1,3,4]` **Output:** `[1,1,2,3,4,4]`

Example 2:

Input: `list1 = []`, `list2 = []` **Output:** `[]`

Example 3:

Input: `list1 = []`, `list2 = [0]` **Output:** `[0]`

Constraints:

- The number of nodes in both lists is in the range `[0, 50]`.
- `-100 <= Node.val <= 100`
- Both `list1` and `list2` are sorted in **non-decreasing** order.

```
/** 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* mergeTwoLists(ListNode* list1, ListNode* list2) {} };
```