**Algorithm: Reversing a Singly Linked List**

**Input:**

A singly linked list L with nodes containing a value and a reference to the next node.

**Output:**

The head of the reversed singly linked list.

**Steps:**

1. **Initialize Pointers**:
   * Create three pointers: prev, current, and next.
   * Set prev to null (it will eventually become the new head of the reversed list).
   * Set current to the head of the list (L).
2. prev = None
3. current = head # head is the starting point of the linked list
4. **Iterate Through the List**:
   * While current is not null, repeat the following steps:
     + Store the next node:
     + next = current.next
     + Reverse the current node's pointer:
     + current.next = prev
     + Move prev and current one step forward:
     + prev = current
     + current = next
5. **Update the Head**:
   * After the loop ends (when current becomes null), set the head of the linked list to prev, as prev now points to the new head of the reversed list.
6. head = prev

**Complete Code Example:**

Here’s a Python-like pseudocode for better understanding:

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def reverse\_linked\_list(head):

prev = None # Step 1: Initialize pointers

current = head

while current is not None: # Step 2: Iterate through the list

next\_node = current.next # Store next node

current.next = prev # Reverse the link

prev = current # Move prev one step forward

current = next\_node # Move current one step forward

head = prev # Step 3: Update the head

return head

**Complexity Analysis:**

* **Time Complexity**: O(n), where n is the number of nodes in the linked list. Each node is processed once.
* **Space Complexity**: O(1), since only a constant amount of space is used for pointers (prev, current, next).

**Conclusion:**

This algorithm efficiently reverses a singly linked list in place with minimal space usage, making it optimal for scenarios where space constraints are a concern.