# Problem Definition: Reverse Linked List

- Given the head of a singly linked list, the goal is to reverse the list and return the new head of 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 between [0, 5000].
- Each node's value lies between -5000 and 5000.

## Objective:

*The task is to reverse a singly linked list in two different ways:*

1. Iteratively.
2. Recursively.

## Iterative Approach:

- The iterative approach uses a loop to reverse the linked list step by step. Here's a breakdown of the procedure:

## Key Concepts:

- **Pointer Manipulation:** The idea is to reverse the direction of each node's next pointer.
- **Three Key Pointers:**
  - ***prev:*** Tracks the previous node (initially None since the new tail will point to None).
  - ***current:*** Points to the current node in the list (initially the head).
  - ***next_node:*** Temporarily stores the next node in the list so that the traversal can continue after modifying pointers.

## Algorithm:

1. **Initialization:**
   - Set prev to None and current to the head of the list.

2. **Loop through the list:**
   - *In each iteration, perform the following steps:*
     - Save the next node using next_node = current.next.
     - Reverse the current node's pointer by setting current.next = prev.
     - Move prev to the current node (prev = current).
     - Move current to the next node (current = next_node).

3. **Termination:**
   - The loop terminates when current becomes None, indicating that the end of the list has been reached.
   - At this point, prev holds the new head of the reversed list.

4. **Return:**
   - Return prev as the new head of the reversed list.

## Time Complexity:

- O(n), where n is the number of nodes in the list. Each node is processed once during the traversal.

## Space Complexity:

- O(1). The reversal is done in-place, and no additional memory is used apart from a few pointers.

## Recursive Approach:

- In the recursive approach, the list is reversed in a top-down manner by breaking down the problem into smaller subproblems. This approach takes advantage of the recursive call stack.

## Key Concepts:

- **Recursion:** The function will call itself with the next node until the base case is reached.
- **Base Case:** If the list is empty or has only one node, return the node itself.
- **Unwinding the Recursion:** As the recursion unwinds, the next pointer of each node is reversed.

## Algorithm:

1. **Base Case:**
   - If the head is None (empty list) or the list has only one node (i.e., head.next == None), return the head since it is already reversed.

2. **Recursive Step:**

- Call the function recursively on the rest of the list (head.next), which will reverse the list starting from the second node.
- *After the recursive call returns, adjust the next node's pointer to point back to the current node:*
  - Set head.next.next = head, reversing the pointer.
  - Set head.next = None to terminate the list properly.

## 3. Return:

- Once all recursive calls have been processed, return the new head of the reversed list (which is returned from the recursive call chain).

## Time Complexity:

- O(n), where n is the number of nodes in the list. Each node is processed once, similar to the iterative approach.

## Space Complexity:

- O(n) due to the recursion stack. Each recursive call adds a frame to the call stack, resulting in space complexity proportional to the number of nodes.

## Comparison of Both Approaches:

1. **Iterative Approach:**

- ***Efficiency:*** Both time and space complexity are optimal (O(n) time, O(1) space).
- ***In-Place Modification:*** Changes the linked list directly using pointer manipulation.
- ***Simplicity:*** Easy to understand and implement using a loop.

2. **Recursive Approach:**

- ***Efficiency:*** Time complexity is the same (O(n)), but space complexity is higher due to the recursion stack (O(n)).
- ***Elegance:*** The recursive approach is more elegant and follows a divide-and-conquer strategy.
- ***Limitations:*** For very large linked lists (near the constraint limits), recursion depth might exceed stack limits, making the iterative approach more practical for such cases.

## Edge Cases:

- **Empty List:** If the input list is empty (head = None), the function should simply return None.
- **Single Node:** If the list contains only one node, the reversed list is the same as the input list.
- **Negative Values:** Ensure the function handles negative values properly, although it doesn't affect the reversal logic.
- **List with Multiple Nodes:** The reversal logic applies consistently for any number of nodes in the list.

## Follow-Up Question:

- The problem also asks if you can implement both the iterative and recursive approaches. In practice, the iterative approach is generally more efficient in terms of space, while the recursive approach is conceptually simpler and easier to understand for smaller lists.

## Conclusion:

- *Reversing a linked list is a fundamental problem in data structures. Both iterative and recursive solutions have their pros and cons:*
- The iterative approach is more space-efficient and practical for large inputs.
- The recursive approach is more elegant and demonstrates a clear conceptual understanding of recursion and linked list manipulation.

Both solutions achieve the same goal: reversing the linked list in O(n) time.