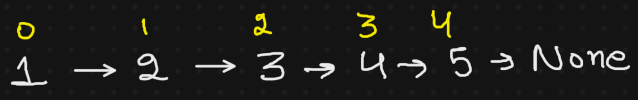
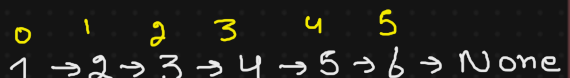
Middle of Linked List



len = 5

middle = len // 2 = 5 // 2 = 2



len = 6

middle = 6 // 2 = 3

→ find length

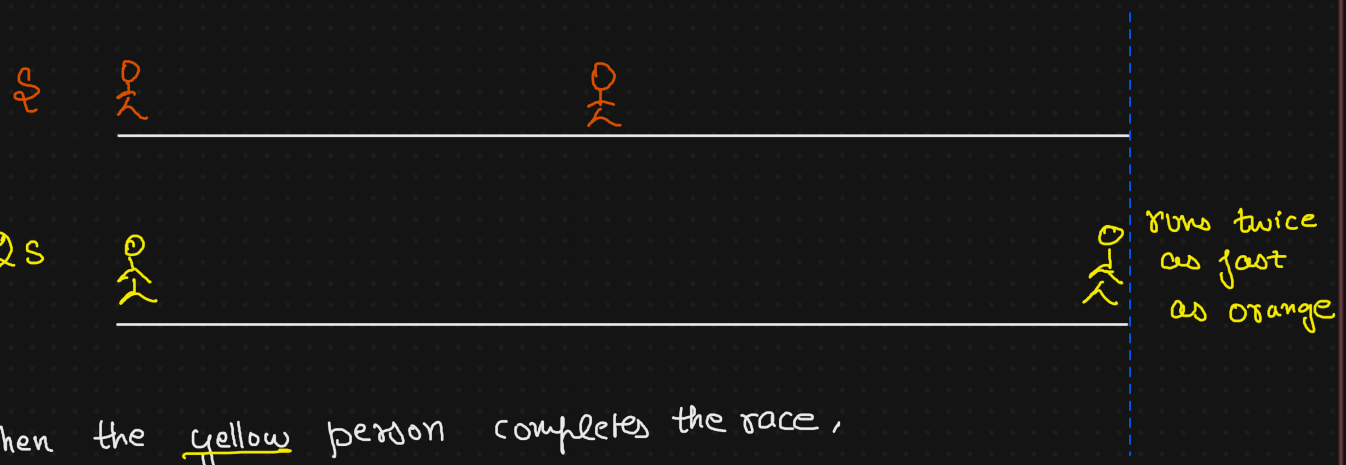
→ find middle

→ temp = head , count = 0

while count < middle

if empty or a single node LL
return head

Middle of LinRed List : 2 pointer approach



When the yellow person completes the race, orange will be at half.

1 → 2 → 3 → 4 → 5 → None

We will move the fast twice while slow moves once

head.next.next.

1st o
y

2nd o
y

3rd o
y

Slow at mid

slow = slow.next

fast = fast.next.next

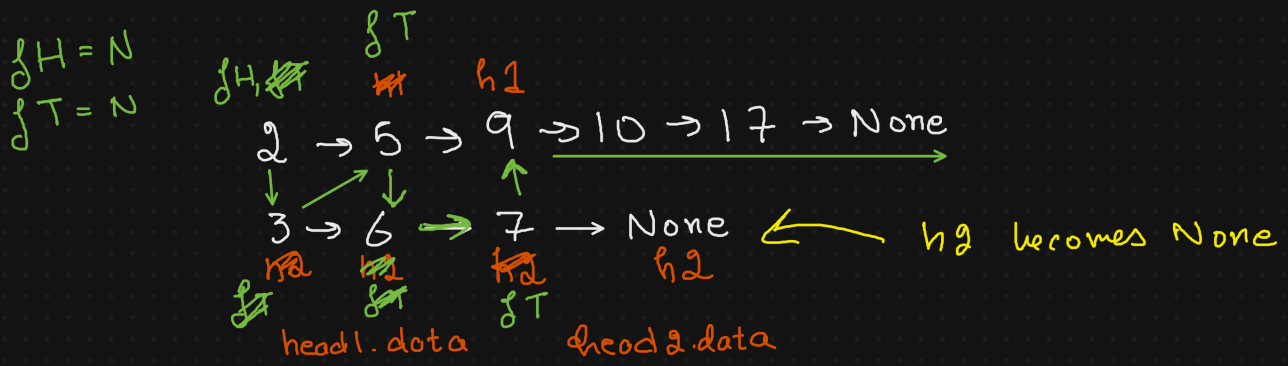
Merge two Sorted Linked List

head 1 : 2 → 5 → 9 → 10 → 17 → None

head 2 : 3 → 6 → 7 → None

Final Head : 2 → 3 → 5 → 6 → 7 → 9 → 10 → 17 → None

Make sure to handle None properly



Reverse a Linked List

head → 1 → 2 → 3 → 4 → 5 → None

Ans 5 → 4 → 3 → 2 → 1 → None

Recursion

1. Base Case — head == None or head.next == None
2. Recursive call — reverse (head.next)
   5 → 4 → 3 → 2 → N
3. Our work — node 1, we just have to connect 1 at tail
   5 → 4 → 3 → 2 → 1 → N

What is the time complexity of our code?

1 → 2 → 3 → 4 → 5 → N

recursive call O(n)

```
def reverse_LL(head):
    # Base Case
    print_LL(head)
    if( head == None or head.next == None ):
        return head

    smallLinkedListHead = reverse_LL(head.next

    temp = smallLinkedListHead
    while(temp.next is not None):
        temp = temp.next

    temp.next = head
    head.next = None

    return smallLinkedListHead
```

finding the tail, it is O(N)

We are doing major work in finding tail

1st node	(n-1) R
2nd	(n-2) R
3rd	(n-3) R
⋮	
last	R
	O(n²)

tail.next

1 → 2 → 3 → 4 → 5 → N

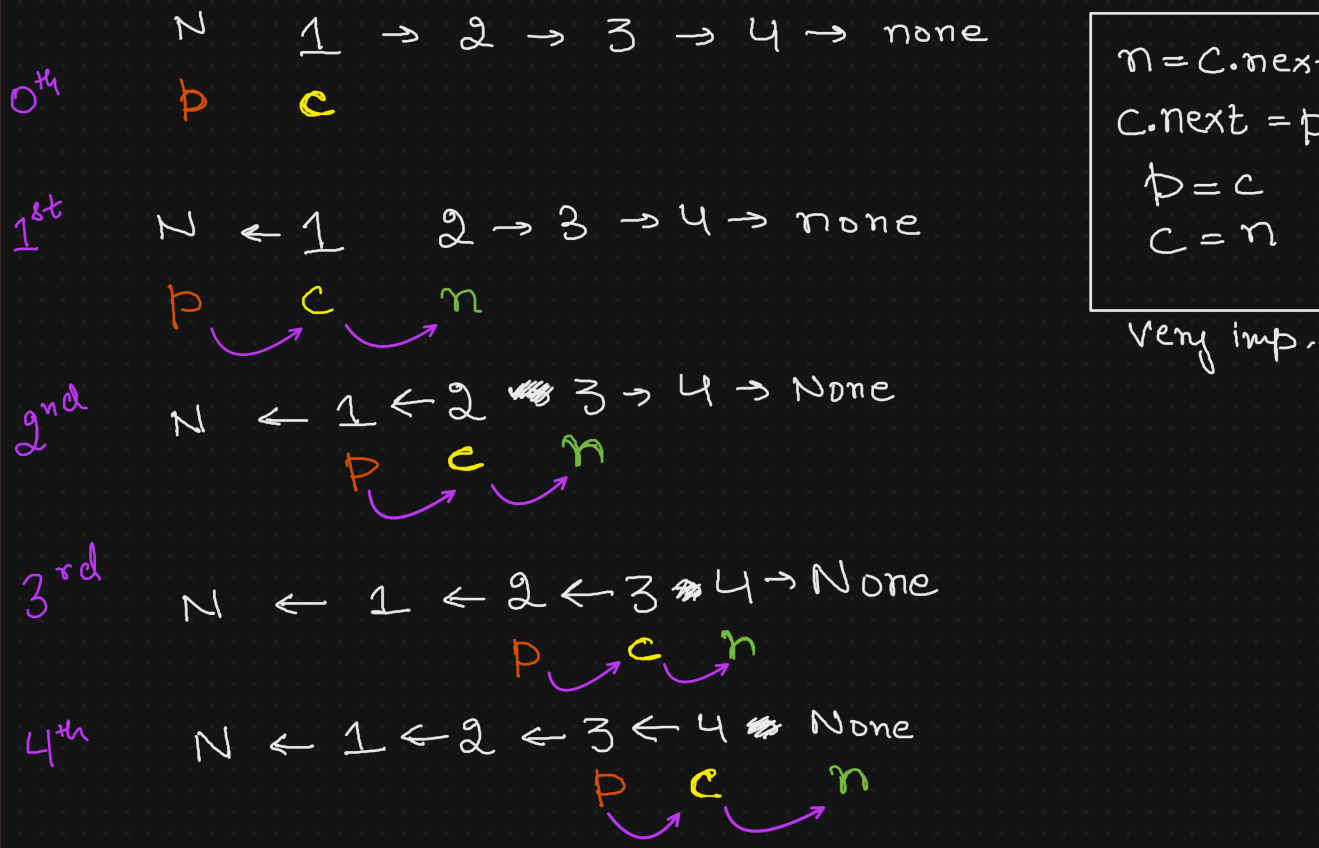
5 → 4 → 3 → 2 → N

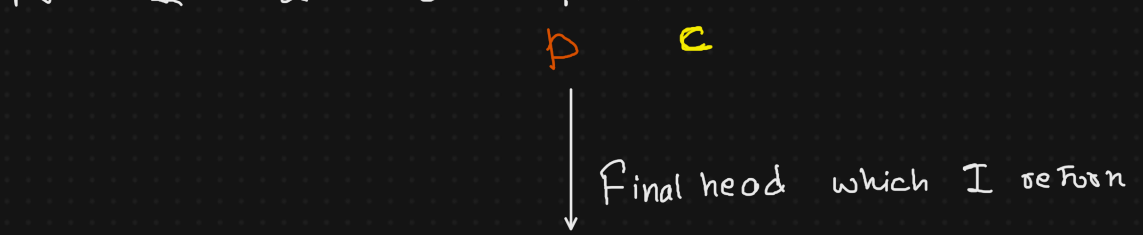
Reversed Linked list (Iteration)

p = None

```
n = c.next
c.next = p
p = c
c = n
```

Very imp.





None ← 1 ← 2 ← 3 ← 4

4 → 3 → 2 → 1 → none