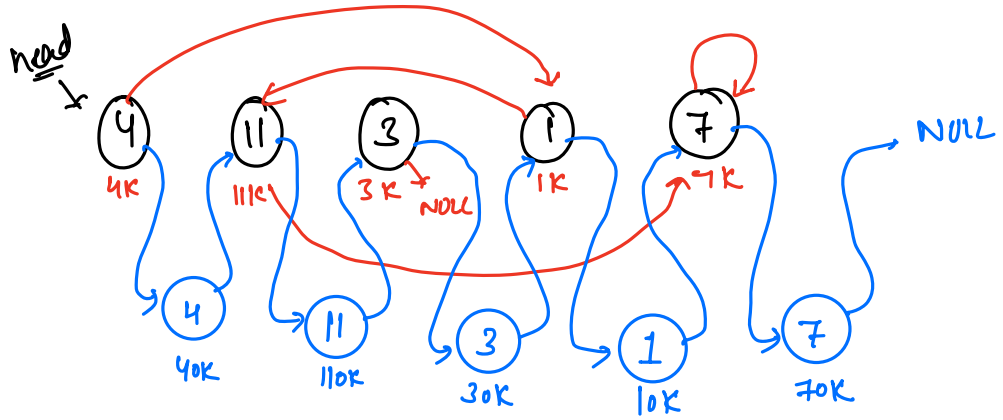
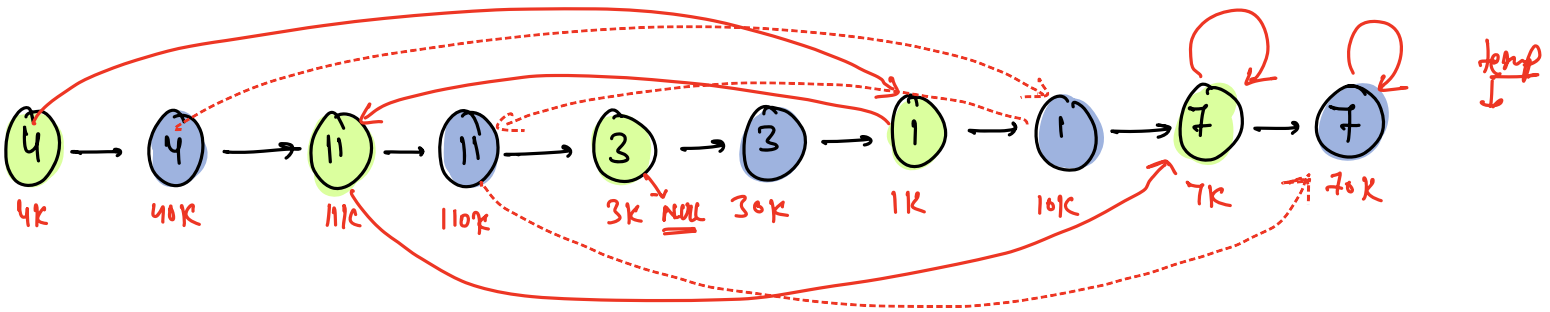
Clone Linked List with Random Pointer





temp.next.random = temp.random.next;

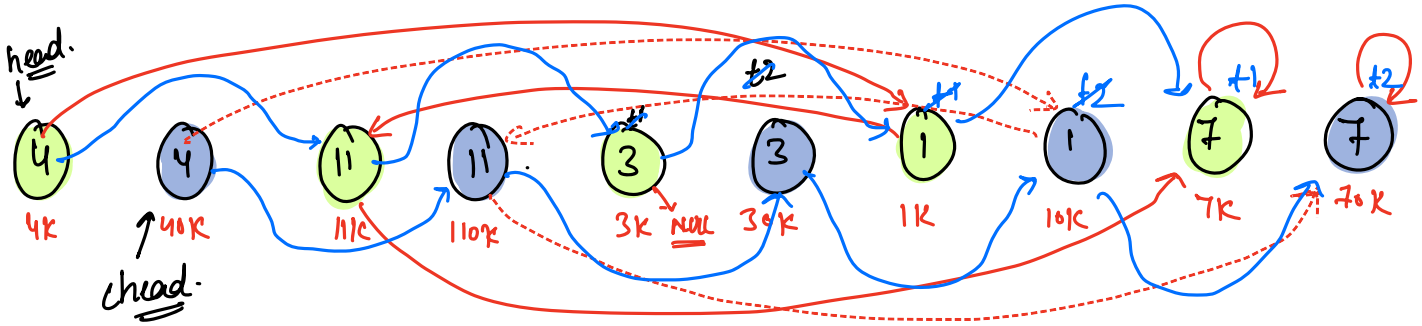
clone

① Insert cloned nodes in the original linked-list.

② Set the random pointers of cloned-nodes.

③ Separate original and cloned linked-list.

Separate ?




```
t1.next = t1.next.next;
t2.next = t2.next.next;
  t1 = t1.next;
  t2 = t2.next;
```

code →

① Insert cloned nodes in the original linked-list.

```
Node temp = head;
while (temp != NULL) {
        nn = new Node(temp.val);
        nn.next = temp.next;
        temp.next = nn;
        //move temp
        temp = temp.next.next;
}
```

② Set the random pointers of cloned-nodes.

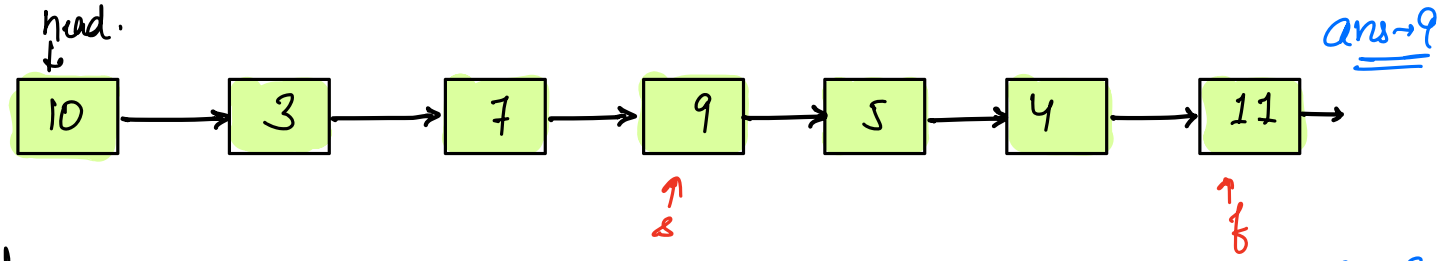
```
temp = head;
while (temp != NULL){
        if( temp.random == NULL){ temp.next.random = NULL) }
    else{ temp.next.random = temp.random.next; }
        temp = temp.next.next;
}
```

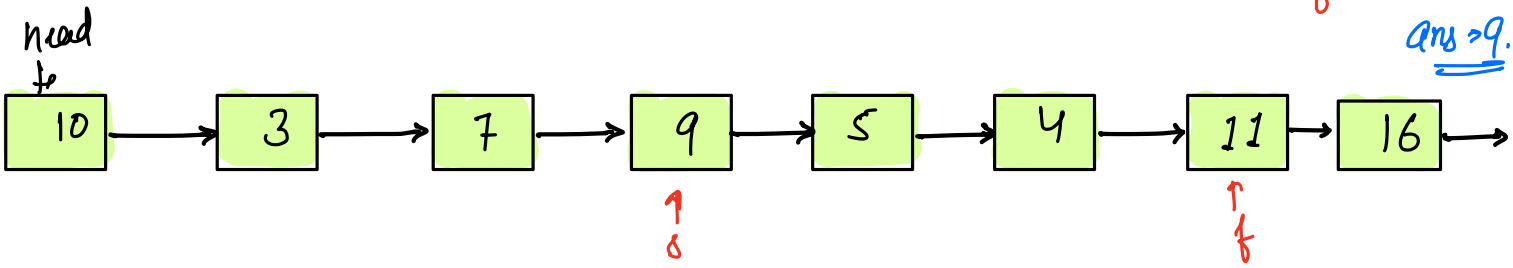
③ Separate original and cloned linked-list

```
chead = head.next;   t1 = head,   t2 = chead;
while (t2.next != NULL){
        t1.next = t1.next.next;
        t2.next = t2.next.next;
        t1 = t1.next;
        t2 = t2.next;
}
t1.next = NULL;
return chead;
```

T.C → O(N)
S.C → O(1)

Middle of A Linked List.



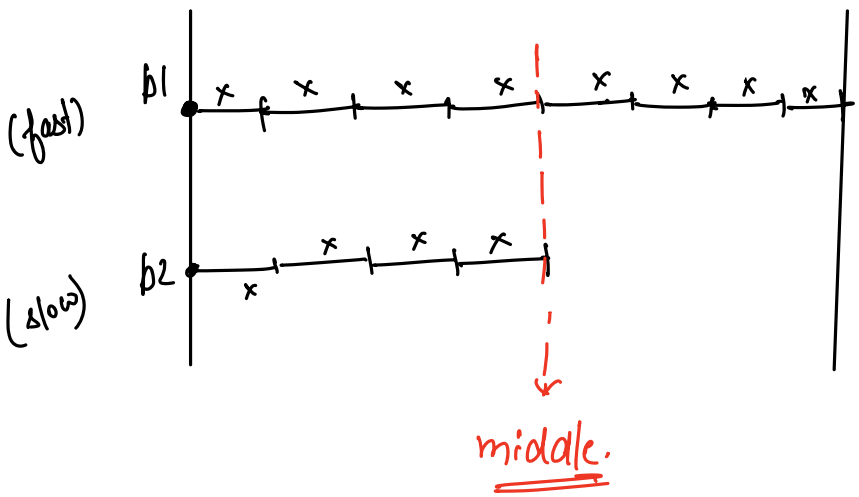


idea-1 → iterate and find the length of the linked list.

ans → element present at $\left(\frac{l-1}{2}\right)^{th}$ index.

[T.C → O(N)
S.C → O(1)]

Hint.



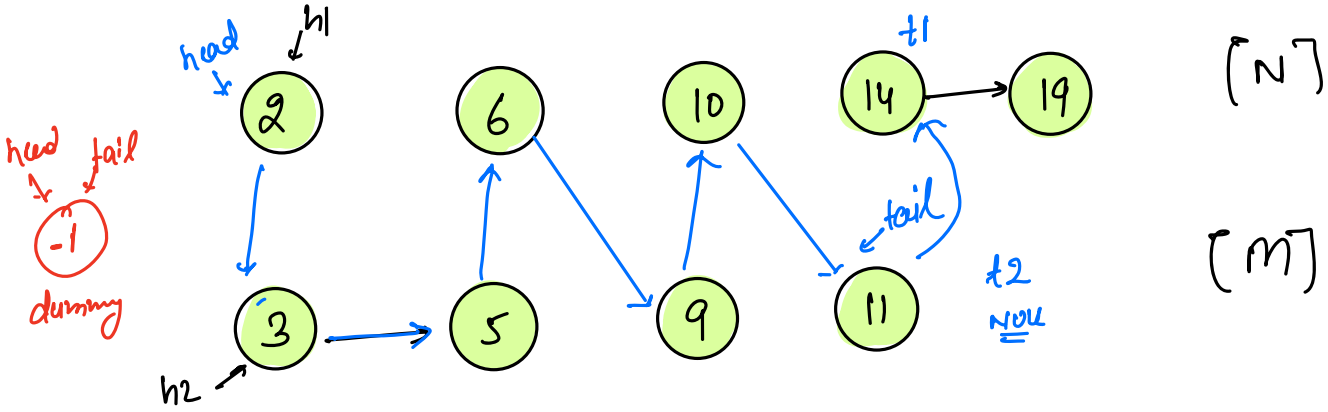
$[s_{p_1} = 2 * s_{p_2}]$

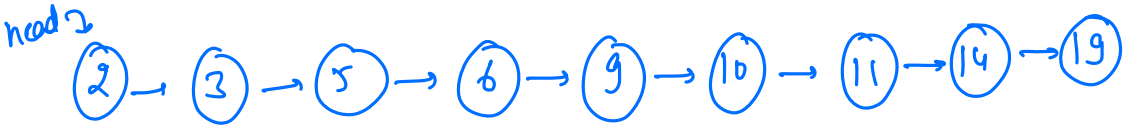
#code →

```
Node slow = head, fast = head;
while( fast.next != NULL && fast.next.next != NULL){
        slow = slow.next;
        fast = fast.next.next;
}
return slow.val;
```

[T.C → O(N)
S.C → O(1)]

Merge Two Sorted Linked-lists





#code...

```
Node head = NULL, tail = NULL, t1 = NULL, t2 = NULL;

if(h1 == NULL){ return h2}
if(h2 == NULL){ return h1}

if(h1.val ≤ h2.val) { head = h1, tail = h1, t1 = h1.next, t2 = h2 }
else { head = h2, tail = h2, t1 = h1, t2 = h2.next }

while( t1 != NULL && t2 != NULL) {
    if(t1.val ≤ t2.val){
        tail.next = t1;
        tail = tail.next;
        t1 = t1.next;
    }
    else{
        tail.next = t2;
        tail = tail.next;
        t2 = t2.next;
    }
}
```

```
Node dummy = new Node(-1)
head = dummy
tail = dummy
t1 = h1;
t2 = h2;
```

```
if (t1 == NULL){
    tail.next = t2;
}
if ( t2 == NULL){
    tail.next = t1;
}
return head;    // dummy.next ;
```

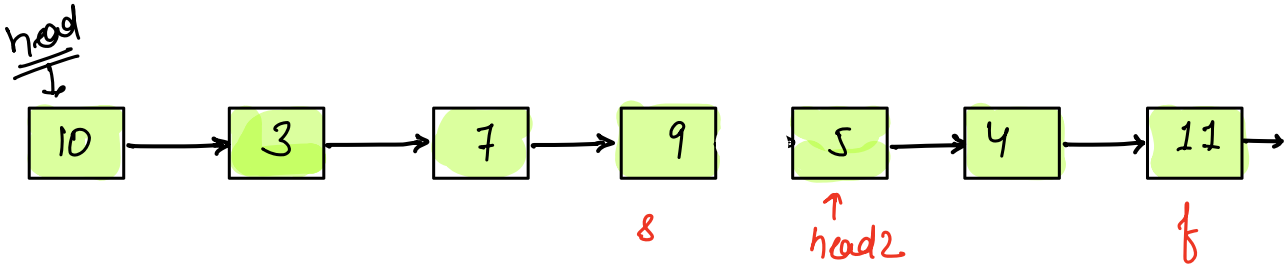
T.C → O(N+m)
S.C → O(1)

→ We can use the concept of dummy node to avoid some edge-cases.

```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, x):
#  self.val = x
#  self.next = None

class Solution:
 # @param A : head node of linked list
 # @return the head node in the linked list
 def sortList(self, A):
        def middle(head): # Getting middle Node of the Linked List Using Slow and Fast Pointers
            slow=head
            fast=head
            while fast.next and fast.next.next:
                slow=slow.next
                fast=fast.next.next
            return slow
        def mergetwoll(l,r): # Merging the two Sorted Linked List
            temp=ans=ListNode(-1)
            while l and r:
                if l.val<=r.val:
                    temp.next=l
                    l=l.next
                else:
                    temp.next=r
                    r=r.next
                temp=temp.next
            while l:
                temp.next=l
                l=l.next
                temp=temp.next
            while r:
                temp.next=r
                r=r.next
                temp=temp.next
            return ans.next
        def solve(head): # Using Merge Sort to Solve the Problem
            if head is None or head.next is None:
                return head
            left=head
            right=middle(head)
            temp=right.next
            right.next=None
            right=temp
            left=solve(left)
            right=solve(right)
            return mergetwoll(left,right)
        return solve(A)
        # Time Complexity--> O(nlogn)
        # Space Complexity--> O(logn)
```

Merge Sort a linked list →



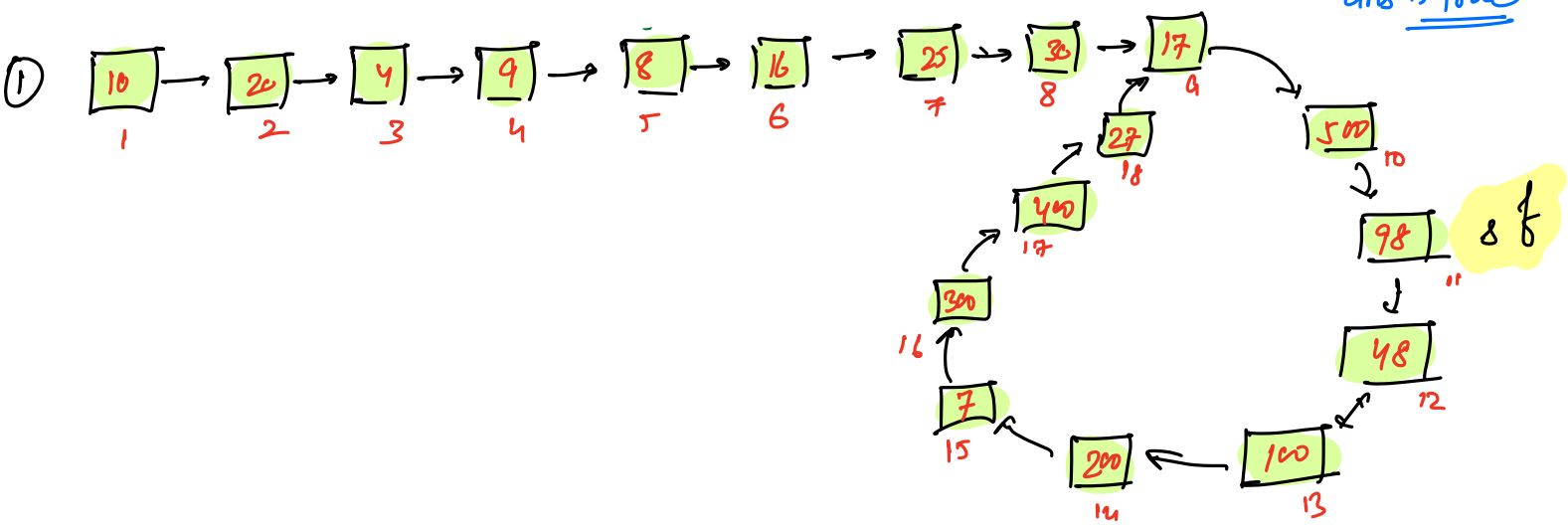

```
Node mergeSortLL ( Node head) {
if(head == NULL || head.next == NULL) {return head}
    mid = getMiddle(head);
    head2 = mid.next;
    mid.next = NULL;
     mergeSortLL(head);
     mergeSortLL ( head2);
    return mergeTwoSortedLL ( head, head2);
}
```

#dry run - todo.

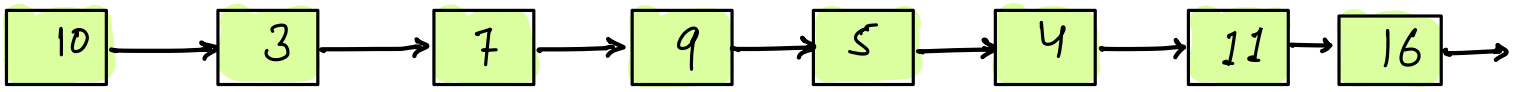
T.C → $O(N \log_2 N)$
S.C → $O(\log_2 N)$

Check if there is a loop

ans → true



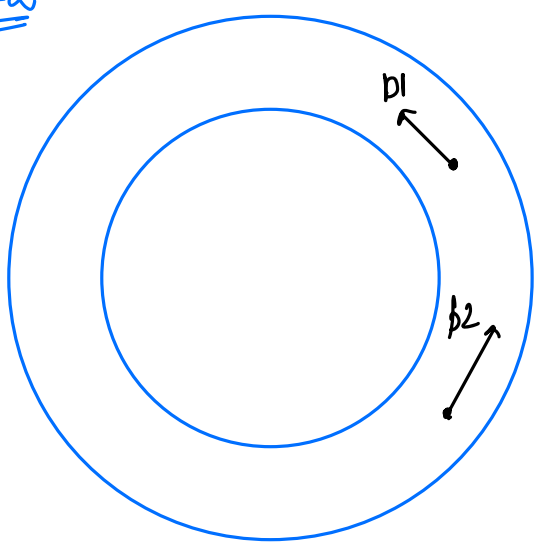
false.



idea → use hashset / hashmap to store node references. Check if any node reference is already present → loop. ✓

[T.C → O(N) , S.C → O(N)]

idea-2



$\&p_1 \neq \&p_2$

⇓

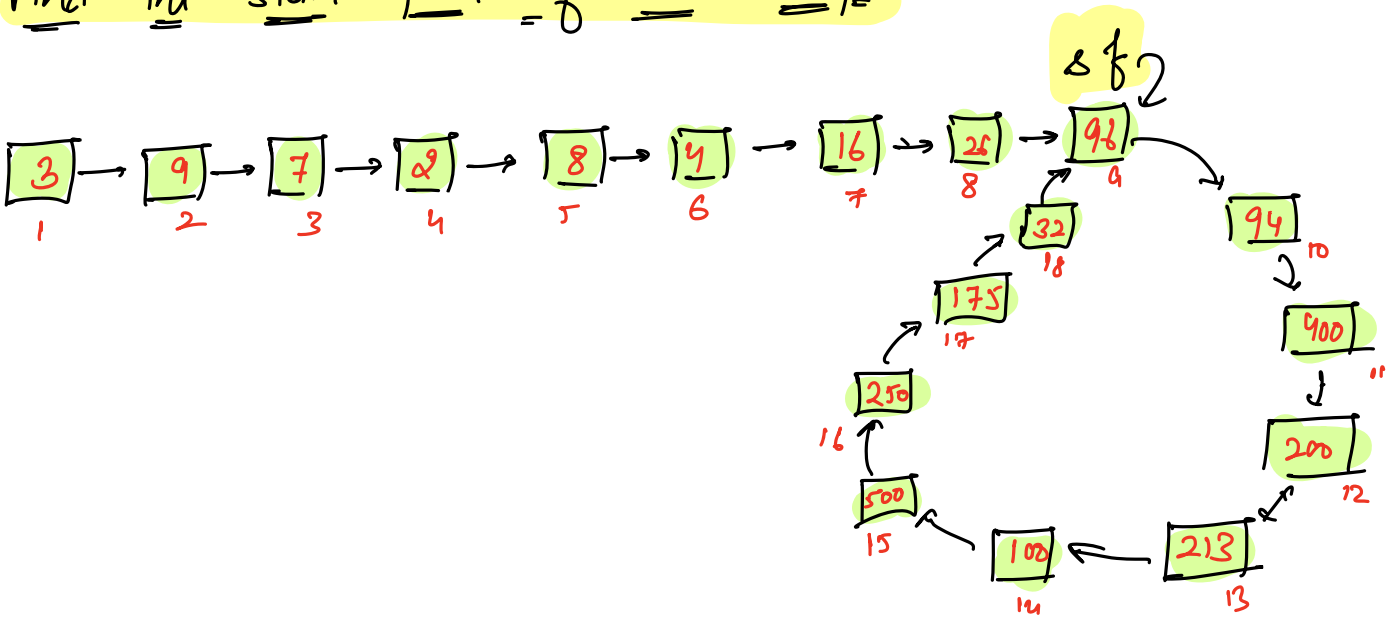
they will definetly meet.

pseudo-code:

```
slow = head, fast = head
while( fast.next != NULL && fast.next.next != NULL){
        slow = slow.next;
        fast = fast.next.next;
        if(slow == fast) { return true; }
}
return false;
```

T.C → O(N)
S.C → O(1)

Find the start point of the loop



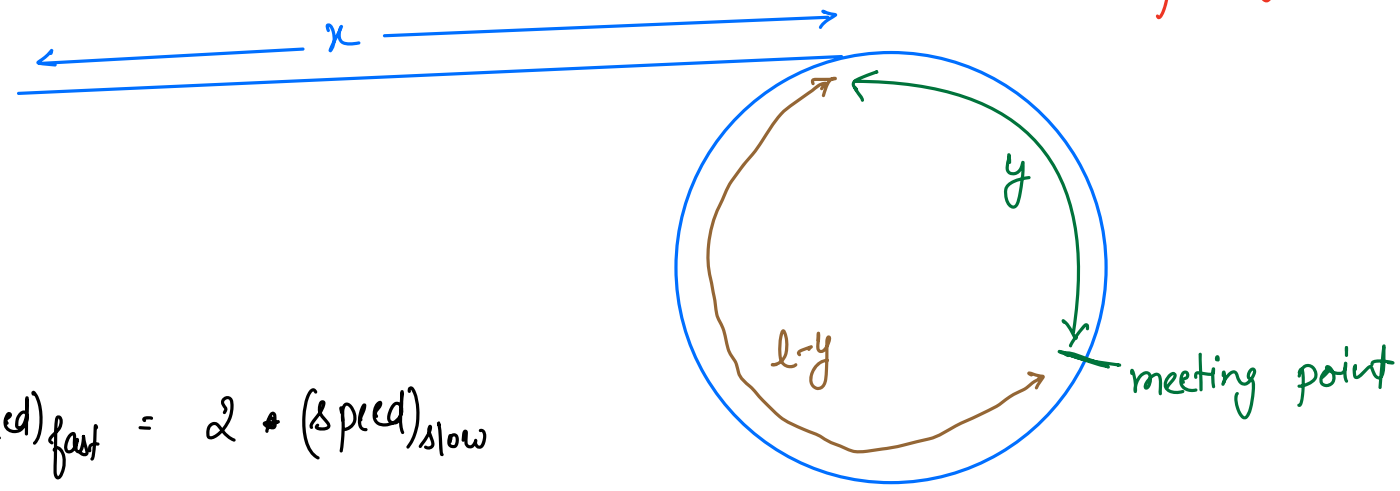
What?

- Reset slow to head
- Move both slow & fast one step at a time simultaneously.

  their meeting point → start point of the loop.

proof →

l → length of loop



$(speed)_{fast} = 2 * (speed)_{slow}$

no. of rotations by slow → R_1

no. of rotations by fast → R_2

distance covered by fast pointer = 2 * distance covered by slow.

$$x + R_2 \cdot l + y = 2 * [x + R_1 \cdot l + y]$$

$$x + R_2 \cdot l + y = 2x + 2R_1 \cdot l + 2y$$

$$R_2 l - 2R_1 \cdot l = 2x + 2y - x - y$$

$$(R_2 - 2R_1) \cdot l = x + y$$

$$R_3 \cdot l = x + y$$

$$\boxed{R_3 \cdot l - y = x}$$

$R_3 = 1 \Rightarrow l - y = x$

$R_3 = 2 \Rightarrow l + (l-y) = x$

$R_3 = 3 \Rightarrow 2l + (l-y) = x$

$\vdots$

$M \cdot l + (l-y) = x$

① Visualisation

② Edge-cases. [dummy]

③ Before submission, dry-run the linked-list problem honestly & check for NPE.

If you are sure about not getting NPE. then only submit it.