Fast and slow pointers

Fast and slow pointers is an implementation of the two pointers technique that we learned with arrays and strings. The idea is to have two pointers that don't move side by side. This could mean they move at different "speeds" during iteration, begin iteration from different locations, or behave differently somehow.

A common case is that the "fast" pointer moves two nodes per iteration whereas the "slow" pointer moves one node per iteration (although this is not always the case). Here is some pseudocode:

```
// head is the head node of a linked list
function fn(head):
    slow = head
    fast = head

    while fast and fast.next:
        Do something here
        slow = slow.next
        fast = fast.next.next
```

The reason we need the while condition to also check for `fast.next` is because if `fast` is at the final node, then `fast.next` is null, and trying to access `fast.next.next` would result in an error (you would be doing `null.next`).

Example 1: Given the head of a linked list with an **odd** number of nodes `head`, return the value of the node in the middle.

One thing we could do is iterate through the linked list once with a dummy pointer to find the length, then iterate from the head again once we know the length to find the middle:

```
def get_middle(head):
    length = 0
    dummy = head
    while dummy:
        length += 1
        dummy = dummy.next
    for _ in range(length // 2):
        head = head.next

    return head.val
```

The most elegant solution comes from using the fast and slow pointer technique. If we have one pointer moving twice as fast as the other, then by the time it reaches the end, the slow pointer will be halfway through since it is moving at half the speed:

```
def get_middle(head):
    slow = head
    fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow.val
```

The pointers use $O(1)$ space, and if there are n nodes in the linked list, the time complexity is $O(n)$ for the traversals.

Example 2: 141 - Linked List Cycle

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

There is a cycle if there is some node in the list that can be reached again by continuously following the `next` pointer.

If a linked list has a cycle, you can imagine some group of nodes forming a circle, and traversal never ends as it moves around that circle infinitely. One way to try to solve this problem would be to just iterate through the list for an arbitrarily large amount of iterations (the given max length of the linked list). If there are no cycles, then we will eventually reach the end of the list. If there is a cycle, then we will never reach an end and after the huge amount of iterations, we can conclude there is a cycle (only if the maximum length is known otherwise we only know that the linked list probably has no cycles).

A better approach is to use a fast and slow pointer. Imagine a straight racetrack (like the one used in the 100m sprint). If two runners of significantly different speeds are racing, then the slower one will never catch up to the faster one. The faster runner finishing the race is like the fast pointer reaching the end of the linked list.

But what if the runners were instead running around a circular racetrack, and needed to complete many laps? In this case, the faster runner will eventually pass (lap) the slower runner.

So move a fast pointer twice the speed of a slow pointer. If they ever meet (except at the start), then we know there must be a cycle. If the fast pointer reaches the end of the linked list, then there isn't a cycle.

Why will the pointers always meet, and the fast pointer won't just "skip" over the slow pointer in the cycle? After looping around the cycle for the first time, if the fast pointer is one position behind, then the pointers will meet on the next iteration. If the fast pointer is two positions behind, then it will be one position behind on the next iteration, This pattern continues - after looping around once, the fast pointer moves exactly one step closer to the slow pointer at each iteration, so it's impossible for it to "skip" over.

```
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        slow = head
        fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True

        return False
```

This approach gives us a time complexity of $O(n)$ and a space complexity of $O(1)$, where `n` is the number of nodes in the linked list.

Example 3: Given the head of a linked list and an integer `k`, return the kth node from the end.

For example, given the linked list that represents `1 -> 2 -> 3 -> 4 -> 5` and `k = 2`, return the node with value `4`, as it is the 2nd node from the end.

If we separate the two pointers by a gap of `k`, and then move them at the same speed, they will always be `k` apart. When the fast pointer (the one further ahead) reaches the end, the slow pointer must be at the desired node, since it is `k` nodes behind.

```
def find_node(head, k):
    slow = head
    fast = head
    for _ in range(k):
        fast = fast.next

    while fast:
        slow = slow.next
        fast = fast.next

    return slow
```

The time complexity of this algorithm is $O(n)$ and the space complexity is $O(1)$, where `n` is the number of nodes in the linked list.

SEE PROBLEMS 876, 83