Problem statement:

Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. List1 may have n nodes before it, Give an algorithm for finding the merging point
Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. List1 may have n nodes before it

Give an algorithm for finding the merging point.

Using Two pointers :

1.Initialize two pointers ptr1 and ptr2 at the head1 and head2.

2.Traverse through the lists,one node at a time.

3.When ptr1 reaches the end of a list, then redirect it to the head2.

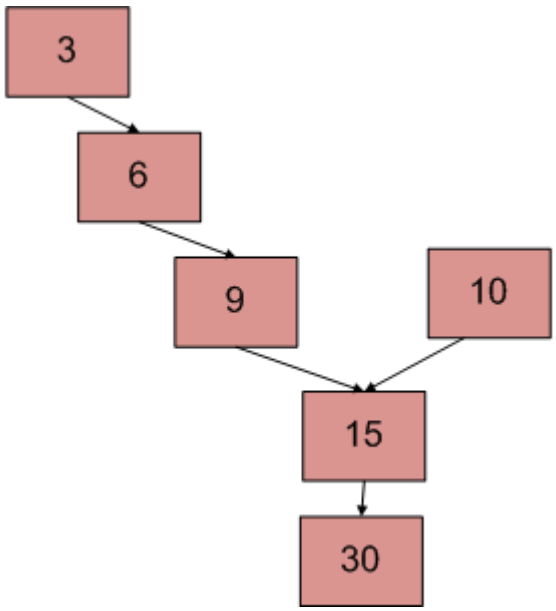
similarly when ptr2 reaches the end of a list, redirect it the head1.

4.Once both of them go through reassigning, they will be equidistant from

the collision point

5.If at any node ptr1 meets ptr2, then it is the intersection node.

6.After second iteration if there is no intersection node it returns NULL.



Can we solve this using the sorting technique?

No it is not possible to solve it using sorting technique.

Can we solve it using hash tables?

Yes it is possible to solve this using hash tables but it is not advisable as it occupies a lot of space.

Can we use stacks for solving?

It is not advisable to use stacks as it takes a lot of time if we use stack.

Is there any other way of solving this ?

Yes, there are many other ways to solve this like: using difference of node counts and by marking the visited nodes.

Can we improve the complexity for?

This code is already good in terms of time complexity, although we can improve it by using doubly linked lists.