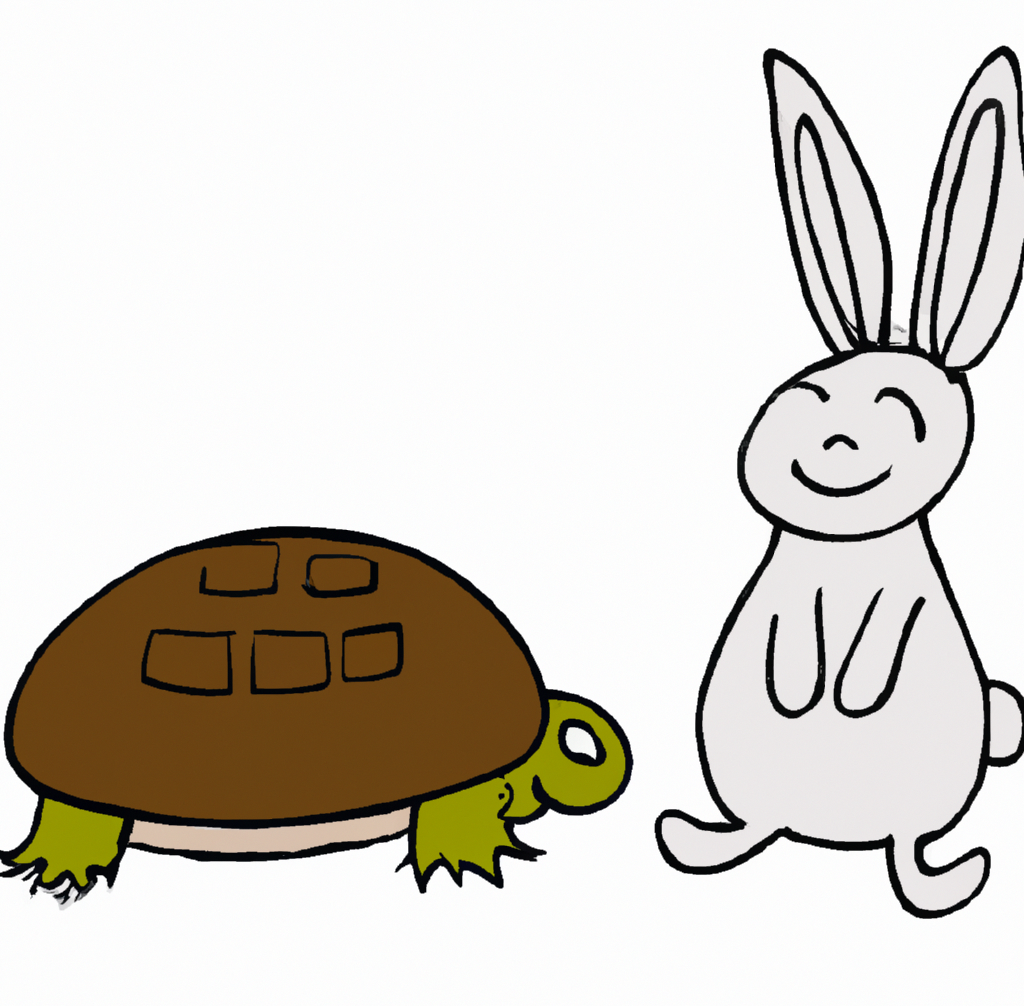
**Floyd's Tortoise and Hare Algorithm**

**Intuition**

This algorithm is very difficult to derive on your own and you would not be expected to do so in an interview without any help.

​

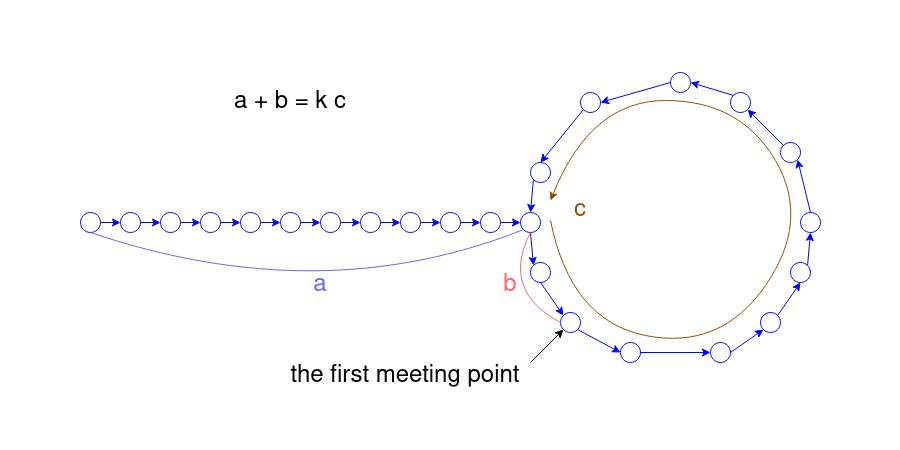
Floyd's Tortoise and Hare Algorithm is a clever technique that is used to detect cycles in sequences or linked lists. You can imagine it as a race between a fast "hare" and a slow "tortoise." We will explain it in a beginner-friendly way:

Imagine you're in a park, where there is a circular path inside the park and a straight path leading to the circular path. If you start walking on the straight path into the circular path, you'll eventually start walking in a cycle around the circular path.

Now imagine two people: a fast runner (the "hare") and a slow walker (the "tortoise"). They both start at the beginning of the path (the start of the linked list). The hare starts running twice as fast as the tortoise.

If the path does not contain a cycle (no circular path), the hare will reach the end of the straight path first. Let's focus on the case where the cycle exists.

At some point, if there is a cycle (a circular path) in the park, the hare will enter this cycle earlier due to its speed. Eventually, the tortoise will also enter the cycle. Since the hare is moving faster, it will lap the tortoise at some point inside the cycle.

​

* Let's define *a* as the length of the path from the start of the list to the entrance of the cycle.
* Let's define *b* as the length of the path from the cycle's entrance to the meeting point of the hare and the tortoise inside the cycle.
* Let's define *c* as the total length of the cycle.

The hare could lap the cycle multiple times before it meets the tortoise, especially if the cycle's size is relatively small compared to the distance from the start to the cycle's entrance, or if the cycle's size is big, and the hare enters it significantly before the tortoise does.

When the tortoise and the hare meet inside the cycle, the tortoise has walked *a*+*b* distance.

On the other hand, the hare, which moves twice as fast, has covered this distance and maybe a few more laps around the cycle. So, the total distance the hare ran is *a*+*b* plus *k*⋅*c*, where *k* is the number of times it lapped the cycle. Because the hare moves twice as fast, this total distance is also equal to 2(*a*+*b*).

If we set these two equal: *a*+*b*+*k*⋅*c*=2(*a*+*b*), we obtain *k*⋅*c*=*a*+*b*.

This tells us that the number of times the hare laps the cycle times the length of the cycle equals the distance from the head of the list to the meeting point.

The question now is where is the entrance to the cycle?

Here is where the second part of the algorithm comes in: after finding a meeting point inside the cycle, you'll leave the tortoise there and move the hare back to the starting point of the park (or the head of the linked list). Then, have both the hare and the tortoise move at the same pace (one step at a time). When they meet again, that meeting point is the entrance to the cycle.

You may ask, "Why is this the entrance to the cycle?" Well, let's consider the distances each has traveled.

The first time that the hare and the tortoise meet within the cycle, we have established that:

* The tortoise has travelled *a*+*b* distance.
* The hare has traveled *a*+*b*+*k*⋅*c* distance, where *k* represents how many times the hare has lapped the cycle.
* Because the hare moves at twice the speed, *a*+*b*+*k*⋅*c*=2(*a*+*b*), rearrange for *k*⋅*c*=*a*+*b*.

If we move the hare back to the start of the straight path and make it move at the same speed as the tortoise, here's what happens:

* The hare has *a* distance to travel to reach the entrance of the cycle. We can rearrange the above equation to say that the hare will reach the entrance of the cycle in *a*=*k*⋅*c*−*b* steps.
* Currently, the tortoise is *b* away from the entrance of the cycle. In *k*⋅*c*−*b* steps, where will the tortoise be? Relative to the entrance of the cycle, the tortoise will be at (*k*⋅*c*−*b*)+*b*=*k*⋅*c*. Because *k* is an integer, *c* is defined as the length of the cycle, and this distance is relative to the entrance of the cycle, the tortoise will be at the entrance!

Because the tortoise and hare are now moving at the same speed, after *k*⋅*c*−*b* steps, they will meet again at the entrance of the cycle. This must be the first time they meet again because the hare has just entered the cycle again for the first time. Therefore, to find the entrance of the cycle, we don't actually need the values of *a*,*b*,*c*,*k*. We can just return the node at which they meet again.

**Algorithm**

1. Initialize the tortoise and hare pointers to the head of the linked list.
2. Move the tortoise one step and the hare two steps at a time until they meet or either hare or hare.next becomes null.
3. If the hare or hare.next pointer is null, it means the hare came to the dead end and we return null as there is no cycle.
4. Reset the hare pointer to the head of the linked list.
5. Move both pointers one step at a time until they meet again. The meeting point is the node where the cycle begins.
6. Return the meeting point node.

**Implementation**

**Complexity Analysis**

Let *n* be the total number of nodes in the linked list.

* Time complexity: *O*(*n*).

The algorithm consists of two phases. In the first phase, we use two pointers (the "hare" and the "tortoise") to traverse the list. The slow pointer (tortoise) will go through the list only once until it meets the hare. Therefore, this phase runs in *O*(*n*) time.

In the second phase, we again have two pointers traversing the list at the same speed until they meet. The maximum distance to be covered in this phase will not be greater than the length of the list (recall that the hare just needs to get back to the entrance of the cycle). So, this phase also runs in *O*(*n*) time.

As a result, the total time complexity of the algorithm is *O*(*n*)+*O*(*n*), which simplifies to *O*(*n*).

* Space complexity: *O*(1).

The space complexity is constant, *O*(1), because we are only using a fixed amount of space to store the slow and fast pointers. No additional space is used that scales with the input size. So the space complexity of the algorithm is *O*(1), which means it uses constant space.