Introduction to Fast and Slow Pointers

Let’s go over the Fast and Slow Pointers pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

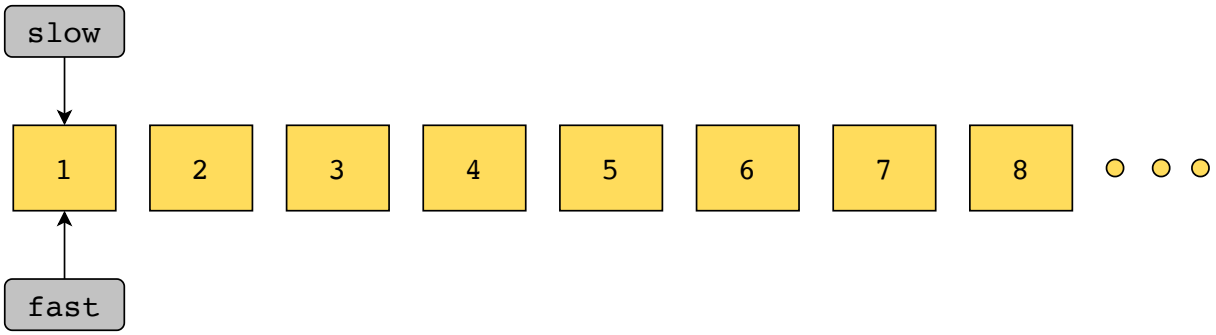
About the pattern

Similar to the two pointers pattern, the **fast and slow pointers** pattern uses two pointers to traverse an iterable data structure, but at different speeds, often to identify patterns, detect cycles, or find specific elements. The speeds of the two pointers can be adjusted according to the problem statement. Unlike the two pointers approach, which is concerned with data values, the fast and slow pointers approach is often used to determine specific pattern or structure in the data.

The key idea is that the pointers start at the same location and then start moving at different speeds. Generally, the slow pointer moves forward by a factor of one, and the fast pointer moves by a factor of two. This approach enables the algorithm to detect patterns or properties within the data structure, such as cycles or intersections. If there is a cycle, the tw are bound to meet at some point during the traversal. To understand the concept, think of two runners on a track. While they start from the same point, they have different running speeds. If the track is circular, the faster runner will overtake the slower one after completing a lap.

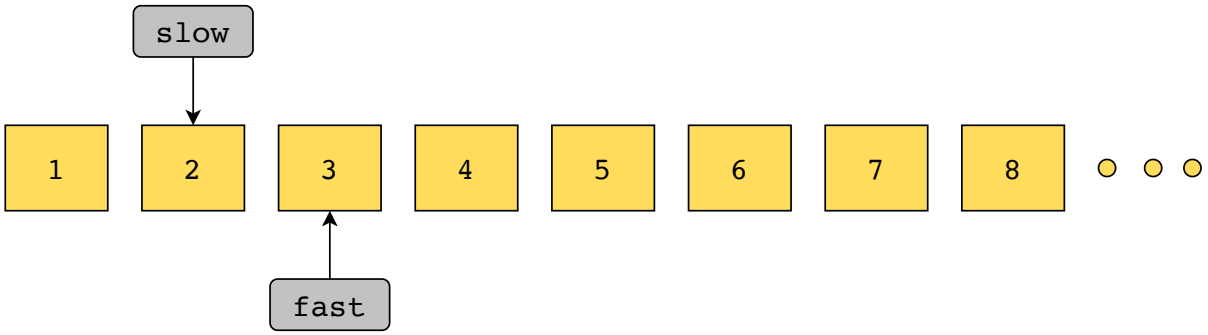
Here's a simple demonstration of how the fast and slow pointers move along a data structure:

Initially, both **slow** and **fast** pointers point to the first position.



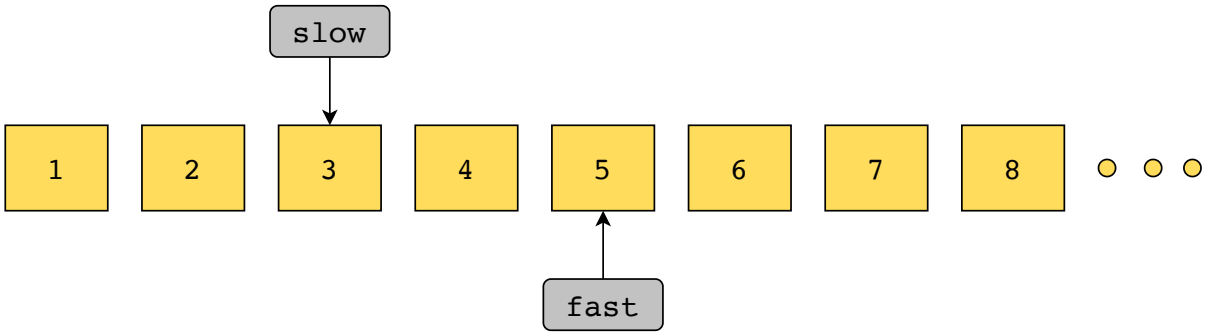
1 of 4

Both pointers move forward with fast pointer moving at double speed as compared to the slow pointer.
Therefore, at first step, the **slow** pointer advances to the second position whereas the **fast** pointer advances to the third position.



2 of 4

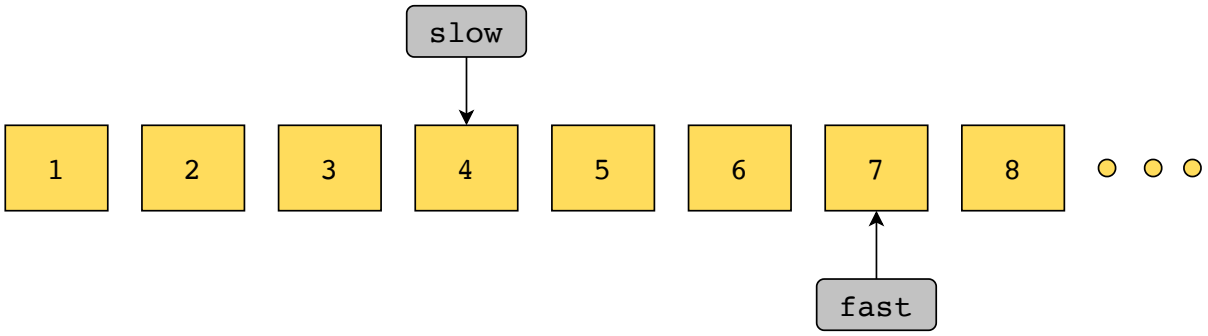
In the second step, the **slow** pointer advances to the third position whereas the **fast** pointer advances to the fifth position.



3 of 4

In the third step, the **slow** pointer advances to the fourth position whereas the **fast** pointer advances to the seventh position.

The **slow** and **fast** pointers keep iterating the data structure in the same manner.



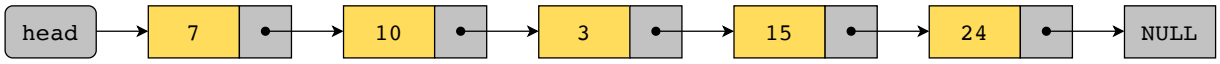
4 of 4

Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Middle of the linked list:** Given the head of a singly linked list, return the middle node of the linked list.

The idea is to iterate the linked using **fast** and **slow** pointers with slow pointer moving one step whereas the fast pointer moving two steps. When the **fast** pointer reaches the last node or becomes NULL, the **slow** pointer will be pointing to the middle of the linked list.



1 of 4

2. **Detect cycle in an array**: Given an array of non-negative integers where elements are indexes pointing to the next element, determine if there is a cycle in the array.

Here is the array in which we need to detect a cycle. The idea is to iterate the array using **fast** and **slow** pointers with **slow** and **fast** pointers moving one step forward and two steps forward, respectively. The pointers move in this way:

- slow = array[slow]
- fast = array[array[fast]]

array	2	3	1	4	0	9	7
indexes	0	1	2	3	4	5	6

Does your problem match this pattern?

Yes, if the following condition is fulfilled:

- **Linear data structure:** The input data can be traversed in a linear fashion, such as an array, linked list, or string.

In addition, if either of these conditions is fulfilled:

- **Cycle or intersection detection:** The problem involves detecting a loop within a linked list or an array or involves finding an intersection between two linked lists or arrays.
- **Find the starting element at the second quantile:** The problem involves finding the starting element of the second quantile, i.e., second half, second tertile, second quartile, etc. For example, the problem asks to find the middle element of an array or a linked list.

Real-world problems

Many problems in the real world use the fast and slow pointers pattern. Let's look at some examples.

- **Symlink verification:** A symbolic link, or symlink, is simply a shortcut to another file. Essentially, it's a file that points to another file. Symlinks can easily create loops or cycles where shortcuts point to each other. To avoid such occurrences, a symlink verification utility can be used, and fast and slow pointers are useful in that case.

- **Compiling an object-oriented program:** Compiling object-oriented programs often involves managing dependencies between various modules stored in separate files for easier maintenance. However, these dependencies can sometimes form cyclic relationships, leading to compilation errors. By employing the fast and slow pointers pattern, these cycles can be detected and resolved, ensuring smooth compilation and execution of the program.

Strategy time!

Match the problems that can be solved using the fast and slow pointers pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Match The Answer

ⓘ Select an option from the left-hand side

Check whether a given string is a palindrome.	Fast and Slow Pointers
Detect a cycle in a linked list.	Some other pattern
Identify if repeatedly computing the sum of	

squares of the digits of number 19 results in 1.

Reverse the words in a given sequence of letters.

Reset Show Solution Submit