**Fast and Slow Pointers: Introduction**

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**

* [Overview](https://www.educative.io/courses/grokking-coding-interview-patterns-java/3YG4vq7M9Bn#Overview)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/3YG4vq7M9Bn#Examples)
* [Does my problem match this pattern?](https://www.educative.io/courses/grokking-coding-interview-patterns-java/3YG4vq7M9Bn#Does-my-problem-match-this-pattern?)
* [Real-world problems](https://www.educative.io/courses/grokking-coding-interview-patterns-java/3YG4vq7M9Bn#Real-world-problems)
* [Strategy time!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/3YG4vq7M9Bn#Strategy-time!)

**Overview**

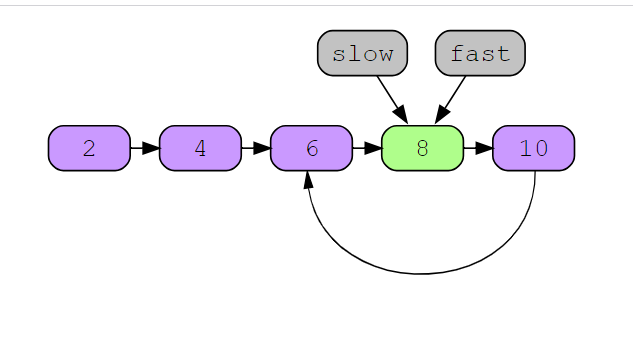
Similar to the two pointers pattern, the **fast and slow pointers** pattern uses two pointers to traverse an iterable data structure at different speeds. It’s usually used to identify distinguishable features of directional data structures, such as a linked list or an array.

The pointers can be used to traverse the array or list in either direction, however, one moves faster than the other. Generally, the slow pointer moves forward by a factor of one, and the fast pointer moves by a factor of two in each step. However, the speed 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 used to determine data structure traits using indices in arrays or node pointers in linked lists. The approach is commonly used to detect cycles in the given data structure, so it’s also known as Floyd’s cycle detection algorithm.

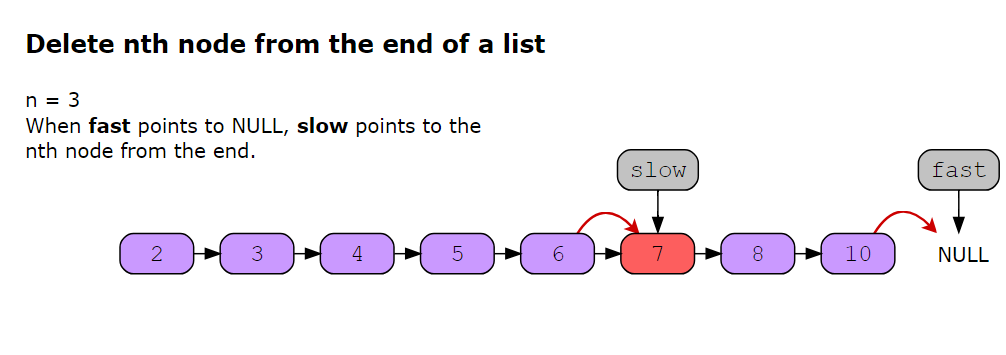
The key idea is that the pointers start at the same location, but they move forward at different speeds. If there is a cycle, the two are bound to meet at some point in the traversal. To understand the concept, think of two runners on a track. While they start from the same point, they have different running speeds. If the race track is a circle, the faster runner will overtake the slower one after completing a lap. On the other hand, if the track is straight, the faster runner will end the race before the slower one, hence never meeting on the track again. The fast and slow pointers pattern uses the same intuition.

Here’s how the pointers move along a list with a cycle:



## Examples

The following example illustrates a problem that can be solved with this approach:



**Does my problem match this pattern?**

* Yes, if either of these conditions is fulfilled:
  + Either as an intermediate step, or as the final solution, the problem requires identifying:
    - the first *x* % of the elements in a linked list, or,
    - the element at the *k*-way point in a linked list, for example, the middle element, or the element at the start of the second quartile, etc.
    - the *kth* last element in a linked list
  + Solving the problem requires detecting the presence of a cycle in a linked list.
  + Solving the problem requires detecting the presence of a cycle in a sequence of symbols.
* No, if either of these conditions is fulfilled:
  + The input data cannot be traversed in a linear fashion, that is, it’s neither in an array, nor in a linked list, nor in a string of characters.
  + The problem can be solved with two pointers traversing an array or a linked list at the same pace.

**Real-world problems**

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

* **Symlink verification:** Fast and slow pointers can be used in a symlink verification utility in an operating system. 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. Similar to a linked list, fast and slow pointers can detect a loop in the symlinks by moving along the connected files or directories at different speeds.
* **Compiling an object-oriented program:** Usually, programs are not contained in a single file. Particularly, for large applications, modules can be divided into different files for better maintenance. Dependency relationships are then defined to specify the order of compilation for these files. However, sometimes, there might be cyclic dependencies that can lead to an error. Fast and slow pointers can be used to identify and remove these cycles for seamless compilation and execution of the program.

**Happy Number**

Try to solve the Happy Number problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/g23V68wZ2vZ#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/g23V68wZ2vZ#Examples)
* [Test your understanding of the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/g23V68wZ2vZ#Test-your-understanding-of-the-problem)
* [Figure it out](https://www.educative.io/courses/grokking-coding-interview-patterns-java/g23V68wZ2vZ#Figure-it-out)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/g23V68wZ2vZ#Try-it-yourself)

**Statement**

Write an algorithm to determine if a number �*n* is a happy number.

We use the following process to check if a given number is a happy number:

* Starting with the given number *n*, replace the number with the sum of the squares of its digits.
* Repeat the process until:
  + The number equals 11, which will depict that the given number *n* is a happy number.
  + It enters a cycle, which will depict that the given number *n* is not a happy number.

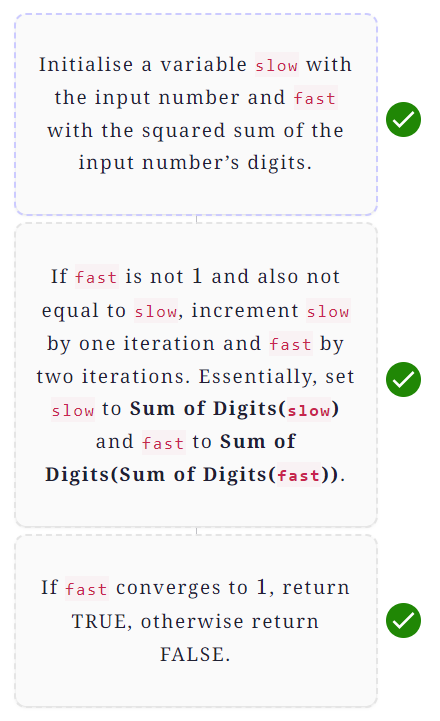
Return TRUE if *n* is a happy number, and FALSE if not.

**Constraints**

* 1≤1≤ *n* ≤231−1≤231−1

## Figure it out

We have a game for you to play: re-arrange the logical building blocks to develop a clearer understanding of how to solve this problem.



# Solution: Happy Number

Let's solve the Happy Number problem using the Fast and slow pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Naive-approach)
  + [Optimized approach using Fast and Slow Pointers pattern](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Optimized-approach-using-Fast-and-Slow-Pointers-pattern)
    - [Step-by-step solution construction](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Step-by-step-solution-construction)
    - [Just the code](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Just-the-code)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Space-complexity)

## Statement

Write an algorithm to determine if a number *n* is a happy number.

We use the following process to check if a given number is a happy number:

* Starting with the given number *n*, replace the number with the sum of the squares of its digits.
* Repeat the process until:
  + The number equals 11, which will depict that the given number *n* is a happy number.
  + It enters a cycle, which will depict that the given number *n* is not a happy number.

Return TRUE if *n* is a happy number, and FALSE if not.

**Constraints**

* 1≤1≤ *n* ≤231−1≤231−1

## Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

The brute force approach is to repeatedly calculate the squared sum of digits of the input number and store the computed sum in a hash set. For every calculation, we check if the sum is already present in the set. If yes, we've detected a cycle and should return FALSE. Otherwise, we add it to our hash set and continue further. If our sum converges to 11, we've found a happy number.

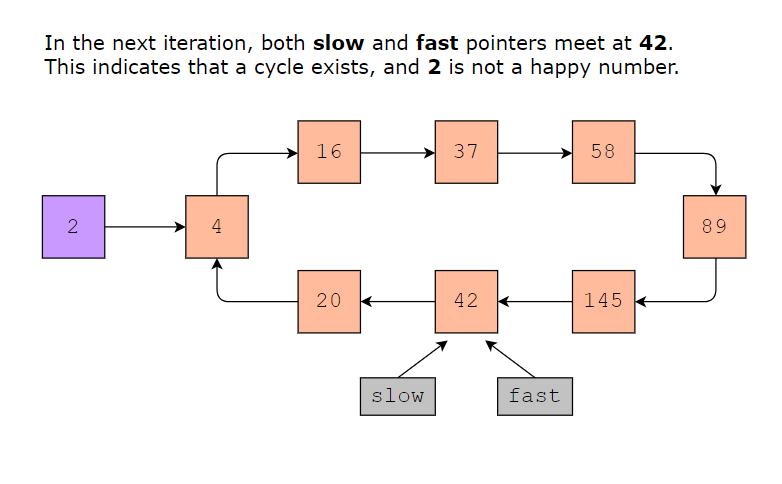
While this approach works well for small numbers, we might have to perform several computations for larger numbers to get the required result. So, it might get infeasible for such cases. The time complexity of this approach is *O*(log*n*). The space complexity is *O*(log*n*) since we're using additional space to store our calculated sums.

### Optimized approach using Fast and Slow Pointers pattern

An efficient approach to solve this problem is to use fast and slow pointers. We know that a unhappy number eventually gets stuck in an infinite loop. However, there is no way for our program to detect this loop and terminate, unless we keep track of the calculated sums, which requires additional space.

If we use the fast and slow pointers approach here, the fast pointer would eventually reach 1, in which case we will return TRUE. Otherwise, it would meet the slow pointer, which would mean that the two pointers are in an endless loop, and we can return FALSE.

As an example, suppose we have the number 22 as our *n*. This is what the infinite loop would look like:



**Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to [just the code](https://www.educative.io/courses/grokking-coding-interview-patterns-java/RLDKqX6OJwY#Just-the-code).

#### Step-by-step solution construction

We will start off our solution by constructing a helper function to calculate the squared sum of digits of the input number. We know that we need to isolate the digits in our number to calculate the squared sum. This can be achieved by repeatedly removing the last digit of the number and adding its squared value to the total sum.

The helper function will find the last digit of the given number by taking its modulus with 1010. We’ll store this in a variable digit. Now, since we’ve already separated the last digit, we can get the remaining digits by dividing the number by 1010. Lastly, we’ll store the squared sum of digit in a variable named totalSum. We’ll repeat this until our number becomes 00.

To understand this better, consider a number, 1919:

**First iteration**

* digit =19%10=9=19%10=9 (last digit)
* number =19/10=1=19/10=1 (remaining digit(s))
* totalSum =92=81=92=81

**Second iteration**

* digit =1%10=1=1%10=1 (last digit)
* number =1/10=0=1/10=0 (remaining digit(s))
* totalSum =81+12=82=81+12=82

As the number has become 00, we’ll terminate our program here. The squared sum of the digits in 1919 is 8282.

class HappyNumber {

public static String printStringWithMarkers(String strn, int pValue) {

String out = "";

for (int i = 0; i < pValue; i++) {

out += String.valueOf(strn.charAt(i));

}

out += "«";

out += String.valueOf(strn.charAt(pValue)) + "»";

for (int i = pValue + 1; i < strn.length(); i++) {

out += String.valueOf(strn.charAt(i));

}

return out;

}

public static int sumOfSquaredDigits(int number) {

int totalSum = 0;

System.out.println("\tCalculating the sum of squared digits");

System.out.println("\tTotal sum: " + totalSum);

int i = 1;

while (number > 0) {

System.out.println("\tLoop iteration: " + i);

int a = String.valueOf(number).length();

System.out.println("\t\tNumber: " + number);

int digit = number % 10;

System.out.println("\t\tWe will start with the last digit of the number " + digit);

System.out.println("\t\t" + printStringWithMarkers(String.valueOf(number), a - 1) + " ⟶ Last Digit: " + digit);

System.out.println("\t\tUpdating number ⟶ number/10 = " + number + "/10");

number = number / 10;

System.out.println("\t\t\t\tThe number is now: " + number);

System.out.println("\t\t\tTotal sum + square of the digit = " + totalSum + " + " + digit + " \* " + digit + " = " + digit);

totalSum += (Math.pow(digit, 2));

i = i + 1;

}

return totalSum;

}

public static void main(String args[]) {

int[] a = {1, 5, 19, 25, 7};

for (int i = 0; i < a.length; i++) {

System.out.println(i + 1 + "." + "\tInput Number:" + a[i]);

System.out.println("\tSum of squared digits: " + sumOfSquaredDigits(a[i]));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

Next, we’ll initialise two variables fast and slow with the input number, and the sum of its digits respectively. In each iteration, we’ll move slow one step forward and fast two steps forward. That is, we’ll call the sumOfSquaredDigits() function once for slow and twice for fast.

* slow == sumOfSquaredDigits(slow)
* fast == sumOfSquaredDigits(sumOfSquaredDigits(fast))

If at any instance fast becomes 11, we’ve found a happy number. We’ll return TRUE in this case. Otherwise, if fast becomes equal to slow, we’ve found a loop and will return FALSE.

class HappyNumber {

public static int sumOfSquaredDigits(int number) {

int totalSum = 0;

while (number != 0) {

int digit = number % 10;

number = number / 10;

totalSum += (Math.pow(digit, 2));

}

System.out.println("\t\tSum of squared digits: " + totalSum);

return totalSum;

}

public static boolean isHappyNumber(int n) {

int slowPointer = n; // The slow pointer value

System.out.println("\tSetting slow pointer = input number " + slowPointer);

System.out.println("\tSetting fast pointer = sum of squared digits of " + n);

int fastPointer = sumOfSquaredDigits(n); // The fast pointer value

System.out.println("\tFast pointer:" + fastPointer);

while (fastPointer != 1 && slowPointer != fastPointer) { // Terminating condition

System.out.println("\n\tRepeatedly updating slow and fast pointers\n");

// Incrementing the slow pointer by 1 iteration

slowPointer = sumOfSquaredDigits(slowPointer);

System.out.println("\tThe updated slow pointer is " + slowPointer);

// Incrementing the fast pointer by 2 iterations

fastPointer = sumOfSquaredDigits(sumOfSquaredDigits(fastPointer));

System.out.println("\tThe updated fast pointer is " + fastPointer + "\n");

}

System.out.println("\tIs it a happy number?: " + (fastPointer == 1)); // If 1 is found then it returns True, otherwise False

return fastPointer == 1;

}

public static void main(String args[]) {

int a[] = {1, 5, 19, 25, 7};

for (int i = 0; i < a.length; i++) {

System.out.println((i + 1) + ".\tInput Number: " + a[i]);

isHappyNumber(a[i]);

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

#### Just the code

Here’s the complete solution to this problem:

class HappyNumber {

public static int sumOfSquaredDigits(int number) {

int totalSum = 0;

while (number != 0) {

int digit = number % 10;

number = number / 10;

totalSum += (Math.pow(digit, 2));

}

return totalSum;

}

public static boolean isHappyNumber(int n) {

int slowPointer = n;

int fastPointer = sumOfSquaredDigits(n);

while (fastPointer != 1 && slowPointer != fastPointer) {

slowPointer = sumOfSquaredDigits(slowPointer);

fastPointer = sumOfSquaredDigits(sumOfSquaredDigits(fastPointer));

}

return fastPointer == 1;

}

public static void main(String args[]) {

int a[] = {1, 5, 19, 25, 7};

for (int i = 0; i < a.length; i++) {

System.out.println((i + 1) + ".\tInput Number: " + a[i]);

String output = isHappyNumber(a[i]) ? "True" : "False";

System.out.println("\n\tIs it a happy number? " + output);

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

#### Solution summary

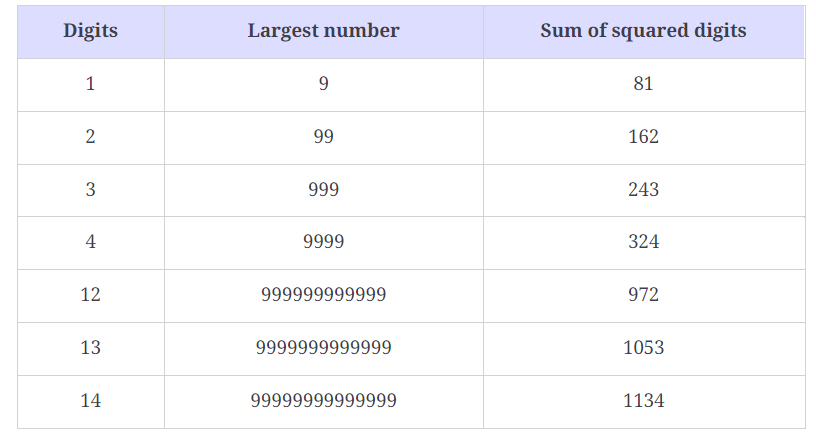
We maintain track of two values using a slow pointer and a fast pointer. The slow runner advances one number at each step, while the fast runner advances two numbers. We detect if there is any cycle by comparing the two values and checking if the fast runner has indeed reached the number one. We return True or False depending on if those conditions are met.

#### Time complexity

The time complexity for this algorithm is *O*(log*n*), where *n* is the input number.

The worst case time complexity of this algorithm is given by the case of a non-happy number, since it gets stuck in a cycle, whereas a happy number quickly converges to 1. Let’s first calculate the time complexity of the **Sum Digits** function. Since we are calculating the sum of all digits in a number, the time complexity of this function is *O*(log*n*), because the number of digits in the number *n* is  log10​*n*.

Before delving into the detailed complexity analysis, let's first consider the largest possible next number for each given number of digits:



To estimate the count of numbers in a cycle, let’s consider the following two cases:

1. **Numbers with three digits:** The largest three-digit number is 999999. The sum of the squares of its digits is 243243. Therefore, for three-digit numbers, no number in the cycle can go beyond 243243. Therefore, the time complexity in this case is given as *O*(243×3), where 243243 is the maximum count of numbers in a cycle and 33 is the number of digits in a three-digit number. This is why the time complexity in the case of numbers with three digits comes out to be *O*(1).
2. **Numbers with more than three digits:** Any number with more than three digits will lose at least one digit at every step until it becomes a three-digit number. For example, the first four-digit number that we can get in the cycle is 10531053, which is the sum of the square of the digits in 99999999999999999999999999. Therefore, the time complexity of any number with more than three digits can be expressed as *O*(log*n*+loglog*n*+logloglog*n*+…). Since *O*(log*n*) is the dominating term, we can write the time complexity as *O*(log*n*).

Therefore, the total time complexity comes out to be *O*(1+log*n*), which is *O*(log*n*).

#### Space complexity

The space complexity for this algorithm is *O*(1).

# Linked List Cycle

Try to solve the Linked List Cycle problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/linked-list-cycle#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/linked-list-cycle#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/linked-list-cycle#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/linked-list-cycle#Figure-it-out)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/linked-list-cycle#Try-it-yourself)

## Statement

Check whether or not a linked list contains a cycle. If a cycle exists, return TRUE. Otherwise, return FALSE. The cycle means that at least one node can be reached again by traversing the next pointer.

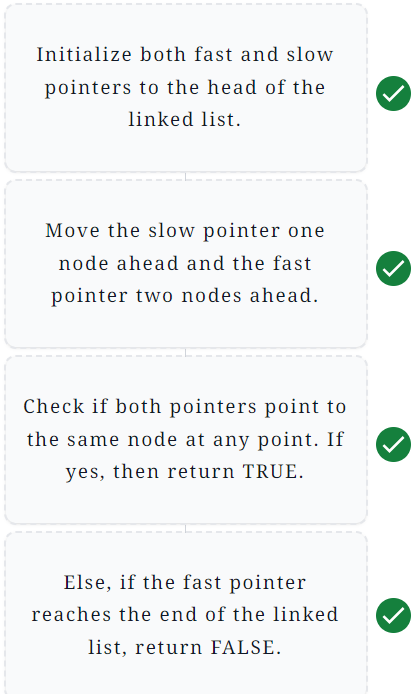
**Constraints:**

Let n be the number of nodes in a linked list.

* 0≤0≤ n ≤500≤500
* −105≤−105≤ Node.data ≤105≤105

## Figure it out![#](https://www.educative.io/courses/grokking-coding-interview-patterns-java/linked-list-cycle#Figure-it-out)

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding of how to solve this problem.



# Solution: Linked List Cycle

Let’s solve the Linked List Cycle problem using the Fast and Slow Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Naive-approach)
  + [Optimized approach using fast and slow pointers](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Optimized-approach-using-fast-and-slow-pointers)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-linked-list-cycle#Space-complexity)

## Statement

Check whether or not a linked list contains a cycle. If a cycle exists, return TRUE. Otherwise, return FALSE. The cycle means that at least one node can be reached again by traversing the next pointer.

**Constraints:**

Let n be the number of nodes in a linked list.

* 0≤0≤ n ≤500≤500
* −105≤−105≤ Node.data ≤105≤105

## Solution

So far, you’ve probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow, based on considerations such as time complexity and any implementation constraints.

### Naive approach

The naive approach is to traverse the linked list and store the current node in a set. At each iteration, we check if the current node is already present in the set. If it is, we’ve found a cycle and return TRUE. Otherwise, we add the node to the set. If the traversal has been completed, return FALSE, since we’ve reached the end of the linked list without encountering a cycle.

The time complexity is *O*(*n*), since we only traverse the linked list once, where *n* represents the total number of nodes in a linked list. The space complexity of the naive approach is also *O*(*n*), since in the worst case, we store *n* nodes in the set.

### Optimized approach using fast and slow pointers

This problem can be solved efficiently using the fast and slow pointers technique, where the fast pointer moves two steps, and the slow pointer moves one step in the linked list.

If there is no cycle in the linked list, the fast pointer will reach the end of the linked list before the slow pointer. If there is a cycle, the fast pointer will eventually catch up to the slow pointer, since it is moving faster. When this happens, we know that we have a cycle in the linked list. The following steps can be performed to implement the algorithm above:

* We initialize two pointers, fast and slow, which point to the head of the linked list.
* We traverse the linked list using these two pointers. They move in the following way:
  + The slow pointer moves only one node forward in each iteration.
  + The fast pointer moves two nodes forward in each iteration, which means that it skips a node.
* If the fast pointer becomes NULL, we have reached the end of the linked list. Since no cycle exists, return FALSE.
* If both slow and fast pointers become equal to each other, return TRUE, since a cycle exists in the linked list.

Here’s a demonstration of the algorithm above:

import java.util.\*;

class CycleDetection {

public static boolean detectCycle(LinkedListNode head) {

if (head == null) {

return false;

}

LinkedListNode slow = head;

LinkedListNode fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

return true;

}

}

return false;

}

// Driver code

public static void main(String args[]) {

int[][] input = { { 2, 4, 6, 8, 10, 12 }, { 1, 3, 5, 7, 9, 11 },

{ 0, 1, 2, 3, 4, 6 }, { 3, 4, 7, 9, 11, 17 }, { 5, 1, 4, 9, 2, 3 } };

int[] pos = { 0, -1, 1, -1, 2 };

for (int i = 0; i < input.length; i++) {

LinkedList<Integer> list = new LinkedList<Integer>();

list.createLinkedList(input[i]);

System.out.print(i + 1 + ".\tInput:");

System.out.print("\t");

if (pos[i] == -1) {

PrintList.printListWithForwardArrow(list.head);

} else {

PrintList.printListWithForwardArrowLoop(list.head);

}

System.out.println("\n\tpos: " + pos[i]);

if (pos[i] != -1) {

int length = list.getLength(list.head);

LinkedListNode lastNode = list.getNode(list.head, length - 1);

lastNode.next = list.getNode(list.head, pos[i]);

}

System.out.println("\n\tDetected Cycle = " + detectCycle(list.head));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

#### Solution summary

To recap, the solution to this problem can be divided into the following three parts:

* Initialize both the slow and fast pointers to the head node.
* Move both pointers at different rates, i.e. the slow pointer will move one step ahead whereas the fast pointer will move two steps.
* If both pointers are equal at some point, we know that a cycle exists.

#### Time complexity

The time complexity of the algorithm is *O*(*n*), where *n* is the number of nodes in the linked list.

#### Space complexity

The space complexity of the algorithm above is *O*(1).

# Middle of the Linked List

Try to solve the Middle of the Linked List problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/middle-of-the-linked-list#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/middle-of-the-linked-list#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/middle-of-the-linked-list#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/middle-of-the-linked-list#Figure-it-out)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/middle-of-the-linked-list#Try-it-yourself)

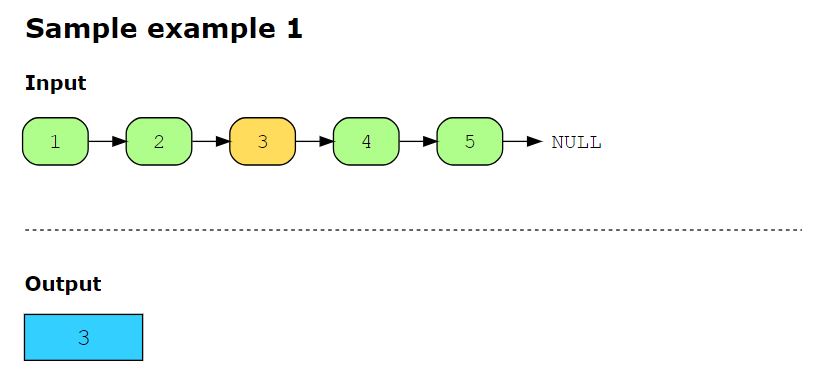
## Statement

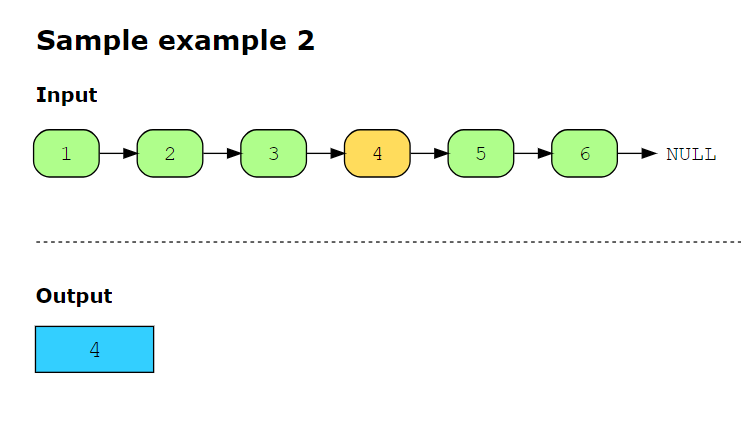
Given the head of a singly linked list, return the middle node of the linked list. If the number of nodes in the linked list is even, there will be two middle nodes, so return the second one.

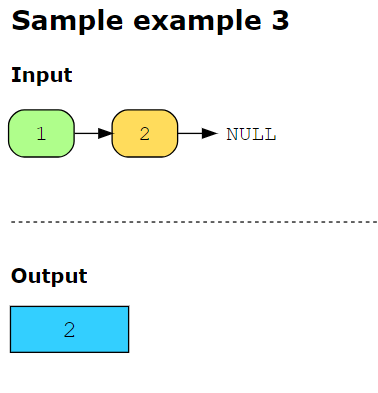
**Constraints:**

Let n be the number of nodes in a linked list.

* 1≤1≤ n ≤100≤100
* 1≤1≤ node.data ≤100≤100
* head ≠= NULL

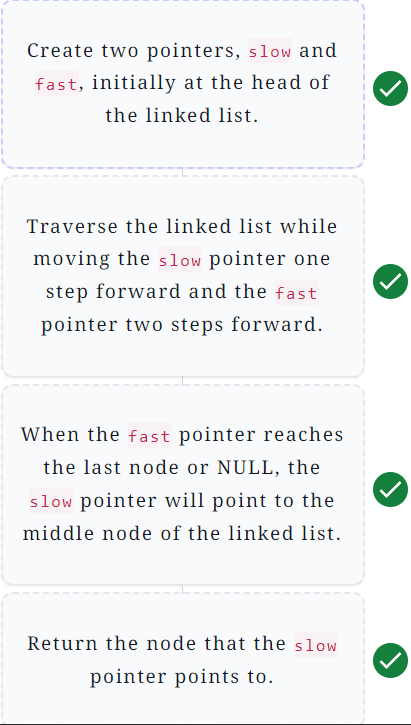






## Figure it out!

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding of how to solve this problem.



# Solution: Middle of the Linked List

Let's solve the Middle of the Linked List problem using the Fast and Slow Pointers pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Naive-approach)
  + [Optimized approach using fast and slow pointers](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Optimized-approach-using-fast-and-slow-pointers)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-middle-of-the-linked-list#Space-complexity)

## Statement

Given the head of a singly linked list, return the middle node of the linked list. If the number of nodes in the linked list is even, there will be two middle nodes, so return the second one.

**Constraints:**

Let n be the number of nodes in a linked list.

* 1≤1≤ n ≤100≤100
* 1≤1≤ node.data ≤100≤100
* head ≠= NULL

## Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

In the naive approach, we use an external array to store the elements of the linked list, and then we return the element present at the index ⌊�����.�����ℎ2⌋⌊2*array*.*length*​⌋ as the middle node of the linked list. The time and space complexity of this approach is �(�)*O*(*n*), where �*n* is the number of nodes in the linked list. Let’s see if we can solve this problem with better time and space complexity.

### Optimized approach using fast and slow pointers

We can use the fast and slow pointers to solve this problem with constant space complexity. The slow pointer traverses the linked list one step at a time, while the fast pointer takes two steps at a time. This makes the fast pointer reach the end of the linked list in �22*n*​ iterations, and the slow pointer, by this time, reaches the middle of the linked list.

The following steps are applied when identifying the middle of the linked list:

1. Create two pointers, slow and fast, initially at the head of the linked list, that is, pointing to the first node.
2. Traverse the linked list using both pointers, where the slow pointer will move one step forward, and the fast pointer will move two steps forward.
3. When the fast pointer reaches the last element of the linked list, or becomes equal to NULL, the slow pointer, at that time, will point to the middle node. Return the node that the slow pointer points to.

The slides below help to understand the solution in a better way.

class MiddleNode {

public static LinkedListNode middleNode(LinkedListNode head) {

LinkedListNode slow = head;

LinkedListNode fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

// Driver code

public static void main( String args[] ) {

int[][] input = {{1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}, {3, 2, 1}, {10}, {1, 2}};

for(int i=0; i<input.length; i++){

System.out.print(i+1);

LinkedList<Integer> list = new LinkedList<Integer>();

list.createLinkedList(input[i]);

System.out.print(".\tInput linked list: ");

PrintList.printListWithForwardArrow(list.head);

System.out.print("\tMiddle of the linked list is: " );

System.out.println(middleNode(list.head).data);

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

#### Solution summary

To recap, the solution to this problem can be divided into the following steps:

1. Create two pointers, slow and fast, initially at the head of the linked list.
2. While traversing the linked list, move the slow pointer one step forward and the fast pointer two steps forward.
3. When the fast pointer reaches the last node or NULL, the slow pointer will point to the middle node of the linked list. Return the node that the slow pointer points to.

#### Time complexity

The time complexity of the solution above is *O*(*n*), where *n* is the number of nodes in the linked list.

#### Space complexity

The space complexity of this solution is constant, that is, *O*(1).