Fast & Slow Pointers

Introduction

The **Fast & Slow** pointer approach, also known as the **Hare & Tortoise algorithm**, is a pointer algorithm that uses two pointers which move through the array (or sequence/LinkedList) at different speeds. This approach is quite useful when dealing with cyclic LinkedLists or arrays.

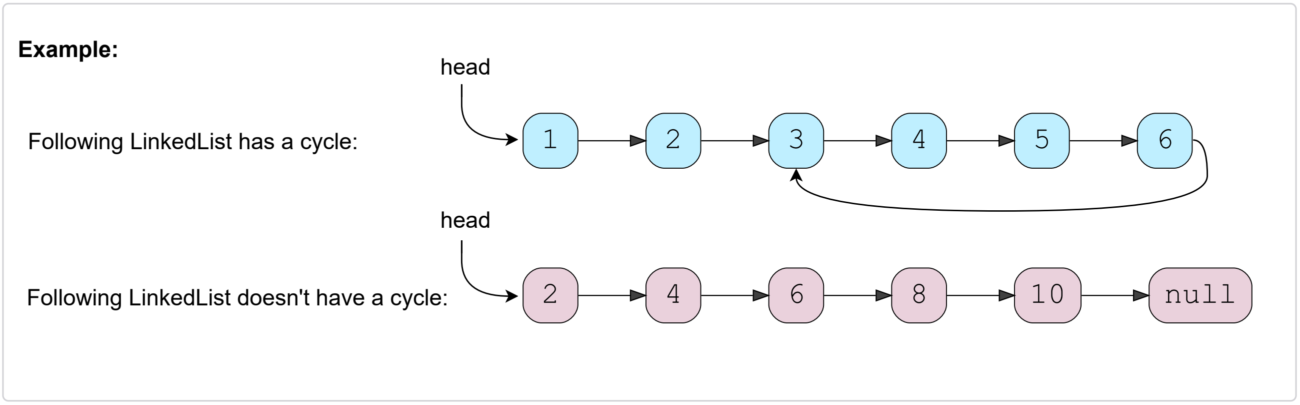
By moving at different speeds (say, in a cyclic LinkedList), the algorithm proves that the two pointers are bound to meet. The fast pointer should catch the slow pointer once both the pointers are in a cyclic loop.

One of the famous problems solved using this technique was **Finding a cycle in a LinkedList**. Let's jump onto this problem to understand the **Fast & Slow** pattern.

LinkedList Cycle (easy)

Problem Statement

Given the head of a **Singly LinkedList**, write a function to determine if the LinkedList has a **cycle** in it or not.


Example:
head
Following LinkedList has a cycle:
1
2
3
4
5
6
head
Following LinkedList doesn't have a cycle:
2
4
6
8
10
null

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycle {
 public:
  static bool hasCycle(ListNode *head) {
    // TODO: Write your code here
    return false;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;

  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;
}
```

Output 3.551s

```
LinkedList has cycle: 0
LinkedList has cycle: 0
LinkedList has cycle: 0
```

Solution

Imagine two racers running in a circular racing track. If one racer is faster than the other, the faster racer is bound to catch up and cross the slower racer from behind. We can use this fact to devise an algorithm to determine if a LinkedList has a cycle in it or not.

Imagine we have a slow and a fast pointer to traverse the LinkedList. In each iteration, the slow pointer moves one step and the fast pointer moves two steps. This gives us two conclusions:

1. If the LinkedList doesn't have a cycle in it, the fast pointer will reach the end of the LinkedList before the slow pointer to reveal that there is no cycle in the LinkedList.
2. The slow pointer will never be able to catch up to the fast pointer if there is no cycle in the LinkedList.

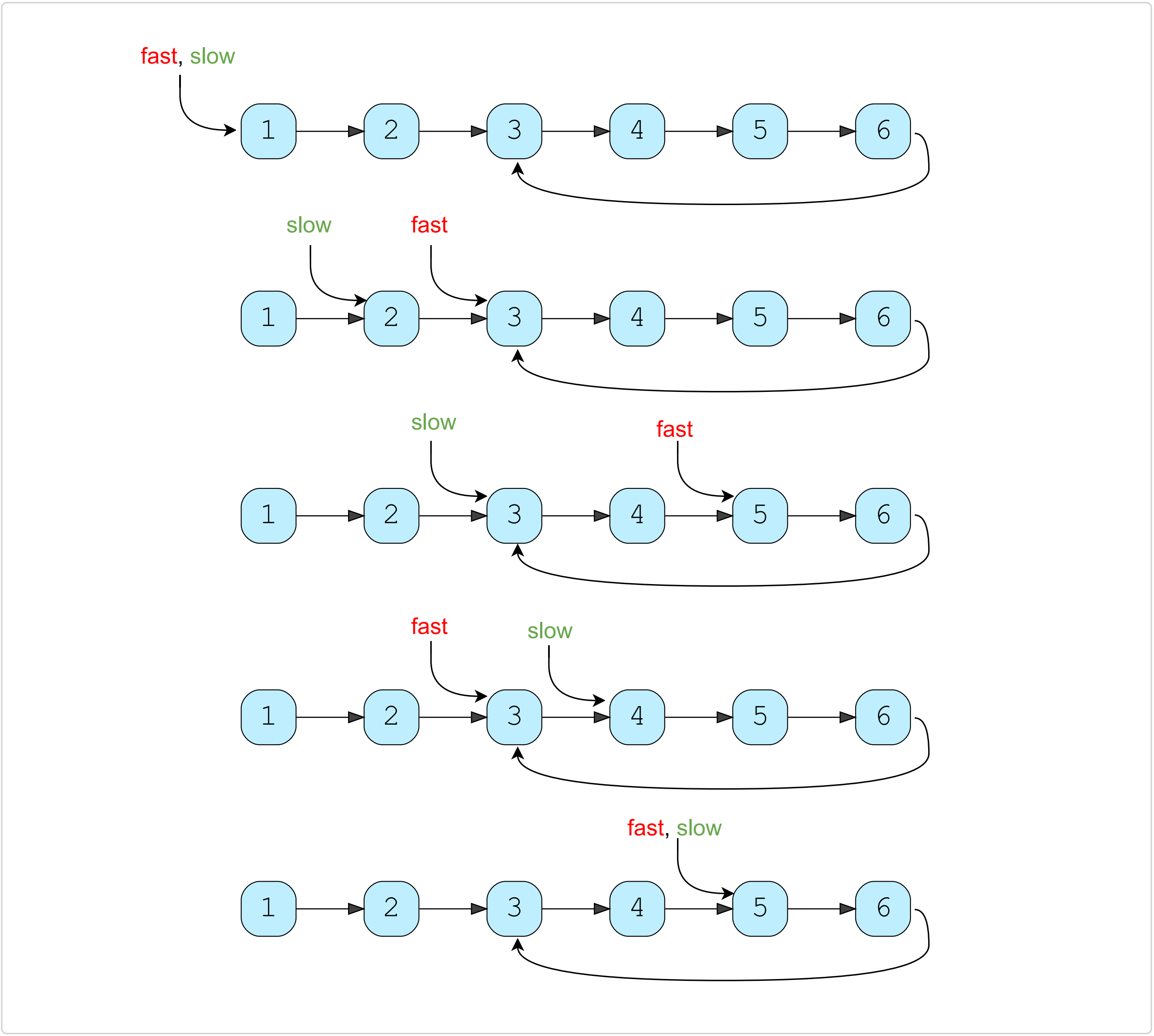
If the LinkedList has a cycle, the fast pointer enters the cycle first, followed by the slow pointer. After this, both pointers will keep moving in the cycle infinitely. If at any stage both of these pointers meet, we can conclude that the LinkedList has a cycle in it. Let's analyze if it is possible for the two pointers to meet. When the fast pointer is approaching the slow pointer from behind we have two possibilities:

1. The fast pointer is one step behind the slow pointer.
2. The fast pointer is two steps behind the slow pointer.

All other distances between the fast and slow pointers will reduce to one of these two possibilities. Let's analyze these scenarios, considering the fast pointer always moves first:

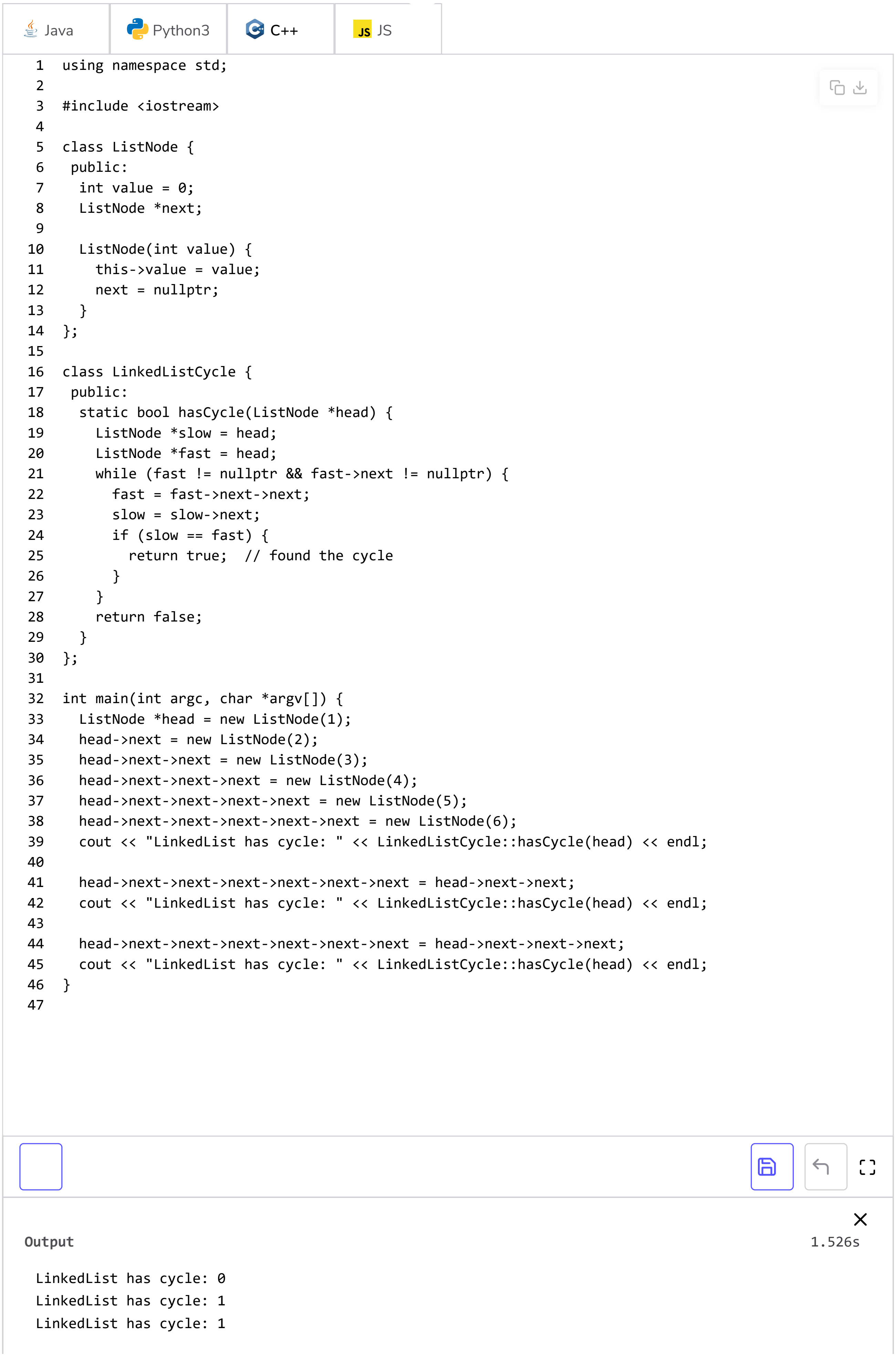
1. **If the fast pointer is one step behind the slow pointer:** The fast pointer moves two steps and the slow pointer moves one step, and they both meet.
2. **If the fast pointer is two steps behind the slow pointer:** The fast pointer moves two steps and the slow pointer moves one step. After the moves, the fast pointer will be one step behind the slow pointer, which reduces this scenario to the first scenario. This means that the two pointers will meet in the next iteration.

This concludes that the two pointers will definitely meet if the LinkedList has a cycle. A similar analysis can be done where the slow pointer moves first. Here is a visual representation of the above discussion:



Code

Here is what our algorithm will look like:



```cpp
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycle {
 public:
  static bool hasCycle(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      fast = fast->next->next;
      slow = slow->next;
      if (slow == fast) {
        return true;  // found the cycle
      }
    }
    return false;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;

  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList has cycle: " << LinkedListCycle::hasCycle(head) << endl;
}
```

Output 1.526s

```
LinkedList has cycle: 0
LinkedList has cycle: 1
LinkedList has cycle: 1
```

Time Complexity

As we have concluded above, once the slow pointer enters the cycle, the fast pointer will meet the slow pointer in the same loop. Therefore, the time complexity of our algorithm will be $O(N)$ where 'N' is the total number of nodes in the LinkedList.

Space Complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

Problem 1: Given the head of a LinkedList with a cycle, find the length of the cycle.

Solution: We can use the above solution to find the cycle in the LinkedList. Once the fast and slow pointers meet, we can save the slow pointer and iterate the whole cycle with another pointer until we see the slow pointer again to find the length of the cycle.

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycleLength {
 public:
  static int findCycleLength(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      fast = fast->next->next;
      slow = slow->next;
      if (slow == fast)  // found the cycle
      {
        return calculateLength(slow);
      }
    }
    return 0;
  }

 private:
  static int calculateLength(ListNode *slow) {
    ListNode *current = slow;
    int cycleLength = 0;
    do {
      current = current->next;
      cycleLength++;
    } while (current != slow);
    return cycleLength;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);
  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList cycle length: " << LinkedListCycleLength::findCycleLength(head) << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList cycle length: " << LinkedListCycleLength::findCycleLength(head) << endl;
}
```

Output 2.586s

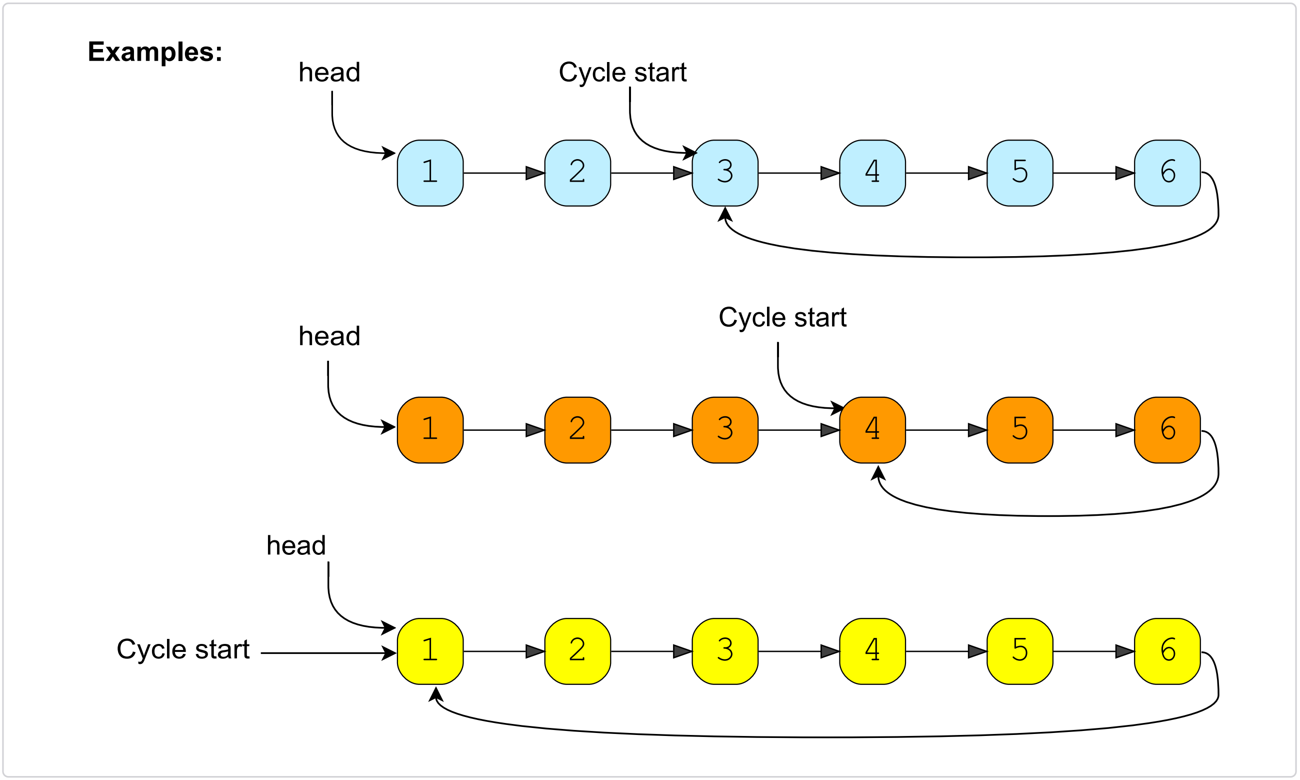
```
LinkedList cycle length: 4
LinkedList cycle length: 3
```

Time and Space Complexity: The above algorithm runs in $O(N)$ time complexity and $O(1)$ space complexity.

Start of LinkedList Cycle (medium)

Problem Statement

Given the head of a **Singly LinkedList** that contains a cycle, write a function to find the **starting node of the cycle**.



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
```

```cpp

class LinkedListCycleStart {
 public:
  static ListNode *findCycleStart(ListNode *head) {
    // TODO: Write your code here
    return head;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);

  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;

  head->next->next->next->next->next->next = head;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;
}

```

Output 1.118s

```
LinkedList cycle start: 1
LinkedList cycle start: 1
LinkedList cycle start: 1
```

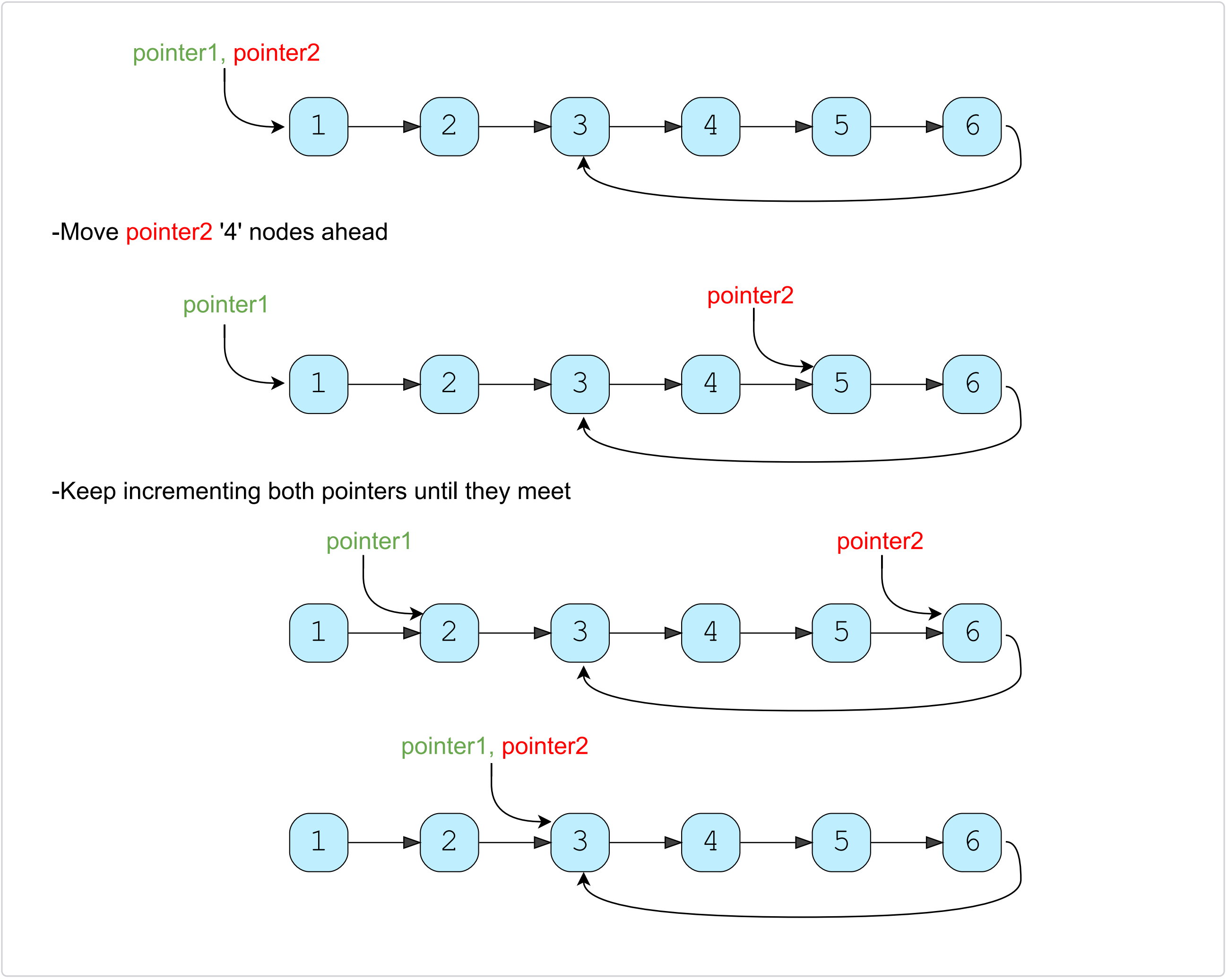
Solution

If we know the length of the **LinkedList** cycle, we can find the start of the cycle through the following steps:

1. Take two pointers. Let's call them `pointer1` and `pointer2`.
2. Initialize both pointers to point to the start of the LinkedList.
3. We can find the length of the LinkedList cycle using the approach discussed in LinkedList Cycle. Let's assume that the length of the cycle is 'K' nodes.
4. Move `pointer2` ahead by 'K' nodes.
5. Now, keep incrementing `pointer1` and `pointer2` until they both meet.

6. As `pointer2` is 'K' nodes ahead of `pointer1`, which means, `pointer2` must have completed one loop in the cycle when both pointers meet. Their meeting point will be the start of the cycle.

Let's visually see this with the above-mentioned Example-1:


pointer1, pointer2
1
2
3
4
5
6
-Move pointer2 '4' nodes ahead
pointer1
pointer2
-Keep incrementing both pointers until they meet
pointer1
pointer2
pointer1, pointer2

We can use the algorithm discussed in LinkedList Cycle to find the length of the cycle and then follow the above-mentioned steps to find the start of the cycle.

Code

Here is what our algorithm will look like:

```cpp
#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class LinkedListCycleStart {
 public:
  static ListNode *findCycleStart(ListNode *head) {
    int cycleLength = 0;
    // find the LinkedList cycle
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      fast = fast->next->next;
      slow = slow->next;
      if (slow == fast) {  // found the cycle
        cycleLength = calculateCycleLength(slow);
        break;
      }
    }

    return findStart(head, cycleLength);
  }

 private:
  static int calculateCycleLength(ListNode *slow) {
    ListNode *current = slow;
    int cycleLength = 0;
    do {
      current = current->next;
      cycleLength++;
    } while (current != slow);

    return cycleLength;
  }

  static ListNode *findStart(ListNode *head, int cycleLength) {
    ListNode *pointer1 = head, *pointer2 = head;
    // move pointer2 ahead 'cycleLength' nodes
    while (cycleLength > 0) {
      pointer2 = pointer2->next;
      cycleLength--;
    }

    // increment both pointers until they meet at the start of the cycle
    while (pointer1 != pointer2) {
      pointer1 = pointer1->next;
      pointer2 = pointer2->next;
    }

    return pointer1;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  head->next->next->next->next->next = new ListNode(6);

  head->next->next->next->next->next->next = head->next->next;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;

  head->next->next->next->next->next->next = head->next->next->next;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;

  head->next->next->next->next->next->next = head;
  cout << "LinkedList cycle start: " << LinkedListCycleStart::findCycleStart(head)->value << endl;
}
```

Output 1.140s

```
LinkedList cycle start: 3
LinkedList cycle start: 4
LinkedList cycle start: 1
```

Time Complexity

As we know, finding the cycle in a LinkedList with ‘N’ nodes and also finding the length of the cycle requires $O(N)$. Also, as we saw in the above algorithm, we will need $O(N)$ to find the start of the cycle. Therefore, the overall time complexity of our algorithm will be $O(N)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

Happy Number (medium)

Problem Statement

Any number will be called a happy number if, after repeatedly replacing it with a number equal to the **sum of the square of all of its digits, leads us to number '1'**. All other (not-happy) numbers will never reach '1'. Instead, they will be stuck in a cycle of numbers which does not include '1'.

Example 1:

```
Input: 23
Output: true (23 is a happy number)
Explanations: Here are the steps to find out that 23 is a happy number:
```

1. $2^2 + 3^2$ = 4 + 9 = 13
2. $1^2 + 3^2$ = 1 + 9 = 10
3. $1^2 + 0^2$ = 1 + 0 = 1

Example 2:

```
Input: 12
Output: false (12 is not a happy number)
Explanations: Here are the steps to find out that 12 is not a happy number:
```

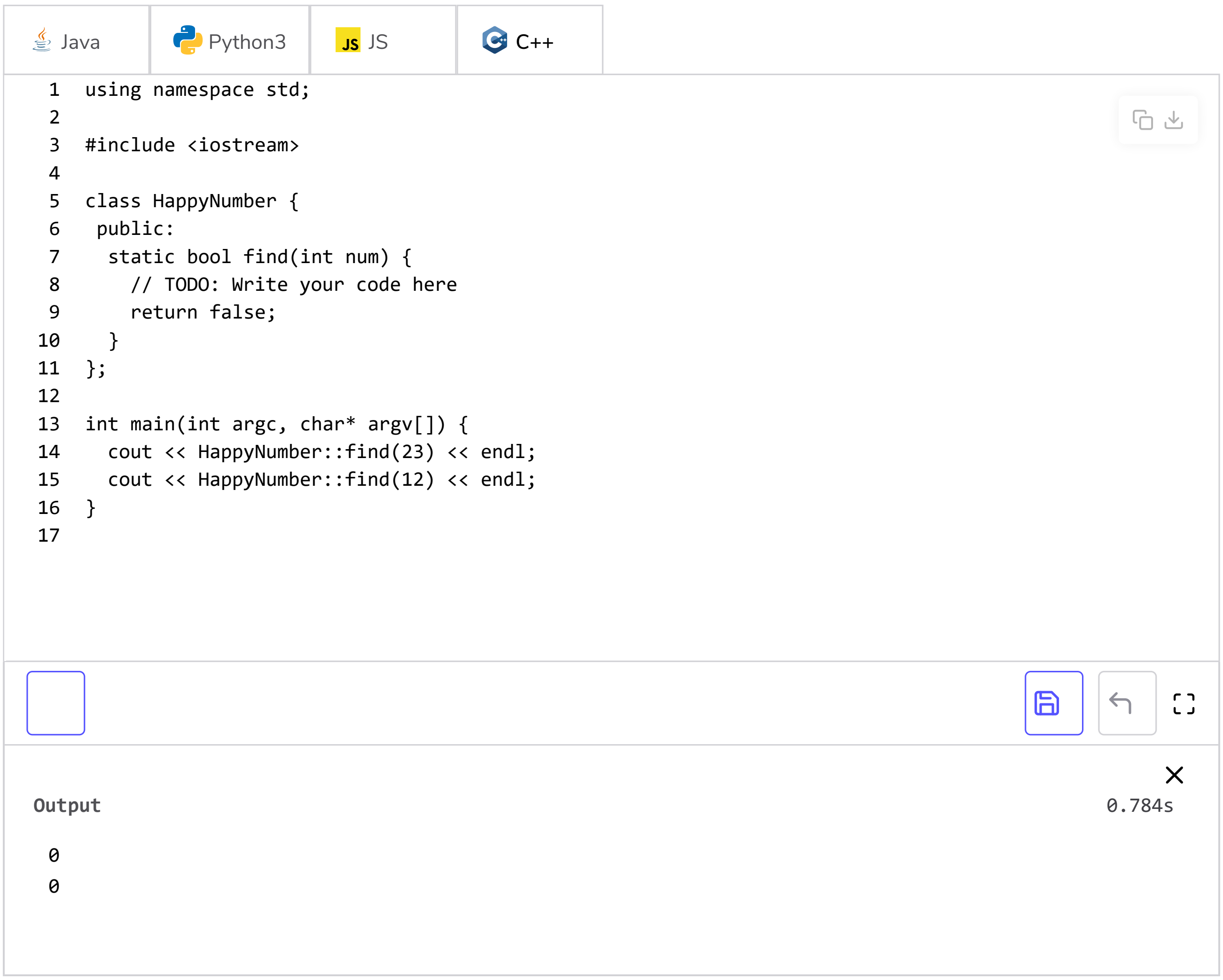
1. $1^2 + 2^2$ = 1 + 4 = 5
2. 5^2 = 25
3. $2^2 + 5^2$ = 4 + 25 = 29
4. $2^2 + 9^2$ = 4 + 81 = 85
5. $8^2 + 5^2$ = 64 + 25 = 89
6. $8^2 + 9^2$ = 64 + 81 = 145
7. $1^2 + 4^2 + 5^2$ = 1 + 16 + 25 = 42
8. $4^2 + 2^2$ = 16 + 4 = 20
9. $2^2 + 0^2$ = 4 + 0 = 4
10. 4^2 = 16

11. $1^2 + 6^2$ = 1 + 36 = 37
12. $3^2 + 7^2$ = 9 + 49 = 58
13. $5^2 + 8^2$ = 25 + 64 = 89

Step ‘13’ leads us back to step ‘5’ as the number becomes equal to ‘89’, this means that we can never reach ‘1’, therefore, ‘12’ is not a happy number.

Try it yourself

Try solving this question here:



```cpp
using namespace std;

#include <iostream>

class HappyNumber {
 public:
  static bool find(int num) {
    // TODO: Write your code here
    return false;
  }
};

int main(int argc, char* argv[]) {
  cout << HappyNumber::find(23) << endl;
  cout << HappyNumber::find(12) << endl;
}

```

```
0
0
```

Solution

The process, defined above, to find out if a number is a happy number or not, always ends in a cycle. If the number is a happy number, the process will be stuck in a cycle on number ‘1,’ and if the number is not a happy number then the process will be stuck in a cycle with a set of numbers. As we saw in Example-2 while determining if ‘12’ is a happy number or not, our process will get stuck in a cycle with the following numbers: 89 -> 145 -> 42 -> 20 -> 4 -> 16 -> 37 -> 58 -> 89

We saw in the LinkedList Cycle problem that we can use the **Fast & Slow pointers** method to find a cycle among a set of elements. As we have described above, each number will definitely have a cycle. Therefore, we will use the same fast & slow pointer strategy to find the cycle and once the cycle is found, we will see if the cycle is stuck on number ‘1’ to find out if the number is happy or not.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>

class HappyNumber {
 public:
  static bool find(int num) {
    int slow = num, fast = num;
    do {
      slow = findSquareSum(slow);                 // move one step
      fast = findSquareSum(findSquareSum(fast));  // move two steps
    } while (slow != fast);                       // found the cycle

    return slow == 1;  // see if the cycle is stuck on the number '1'
  }

 private:
  static int findSquareSum(int num) {
    int sum = 0, digit;
    while (num > 0) {
      digit = num % 10;
      sum += digit * digit;
      num /= 10;
    }
    return sum;
  }
};

int main(int argc, char* argv[]) {
  cout << HappyNumber::find(23) << endl;
  cout << HappyNumber::find(12) << endl;
}
```

```
1
0
```

Time Complexity

The time complexity of the algorithm is difficult to determine. However we know the fact that all unhappy numbers eventually get stuck in the cycle: 4 -> 16 -> 37 -> 58 -> 89 -> 145 -> 42 -> 20 -> 4

This sequence behavior tells us two things:

1. If the number N is less than or equal to 1000, then we reach the cycle or ‘1’ in at most 1001 steps.
2. For $N > 1000$, suppose the number has ‘M’ digits and the next number is ‘N1’. From the above Wikipedia link, we know that the sum of the squares of the digits of ‘N’ is at most 9^2M, or $81M$ (this will happen when all digits of ‘N’ are ‘9’).

This means:

1. $N1 < 81M$
2. As we know $M = log(N+1)$
3. Therefore: $N1 < 81 * log(N+1)$ => $N1 = O(logN)$

This concludes that the above algorithm will have a time complexity of $O(logN)$.

Space Complexity

The algorithm runs in constant space $O(1)$.

Middle of the LinkedList (easy)

Problem Statement

Given the head of a **Singly LinkedList**, write a method to return the **middle node** of the LinkedList.

If the total number of nodes in the LinkedList is even, return the second middle node.

Example 1:

```
Input: 1 -> 2 -> 3 -> 4 -> 5 -> null
Output: 3
```

Example 2:

```
Input: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null
Output: 4
```

Example 3:

```
Input: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> null
Output: 4
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class MiddleOfLinkedList {
  public :
    static ListNode *findMiddle ( ListNode *head ) {
```

```
    // TODO: Write your code here
    return head;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;

  head->next->next->next->next->next = new ListNode(6);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;

  head->next->next->next->next->next->next = new ListNode(7);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;
}

```

Output 0.782s

```
Middle Node: 1
Middle Node: 1
Middle Node: 1
```

Solution

One brute force strategy could be to first count the number of nodes in the LinkedList and then find the middle node in the second iteration. Can we do this in one iteration?

We can use the **Fast & Slow pointers** method such that the fast pointer is always twice the nodes ahead of the slow pointer. This way, when the fast pointer reaches the end of the LinkedList, the slow pointer will be pointing at the middle node.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
using namespace std;

#include <iostream>

class ListNode {
 public :
   int value = 0 ;
   ListNode *next ;

```

```cpp
  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class MiddleOfLinkedList {
 public:
  static ListNode *findMiddle(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
    }

    return slow;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(1);
  head->next = new ListNode(2);
  head->next->next = new ListNode(3);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(5);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;

  head->next->next->next->next->next = new ListNode(6);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;

  head->next->next->next->next->next->next = new ListNode(7);
  cout << "Middle Node: " << MiddleOfLinkedList::findMiddle(head)->value << endl;
}
```

Output 0.907s

```
Middle Node: 3
Middle Node: 4
Middle Node: 4
```

Time complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Challenge 1

Palindrome LinkedList (medium)

Given the head of a **Singly LinkedList**, write a method to check if the **LinkedList is a palindrome** or not.

Your algorithm should use **constant space** and the input LinkedList should be in the original form once the algorithm is finished. The algorithm should have $O(N)$ time complexity where 'N' is the number of nodes in the LinkedList.

Example 1:

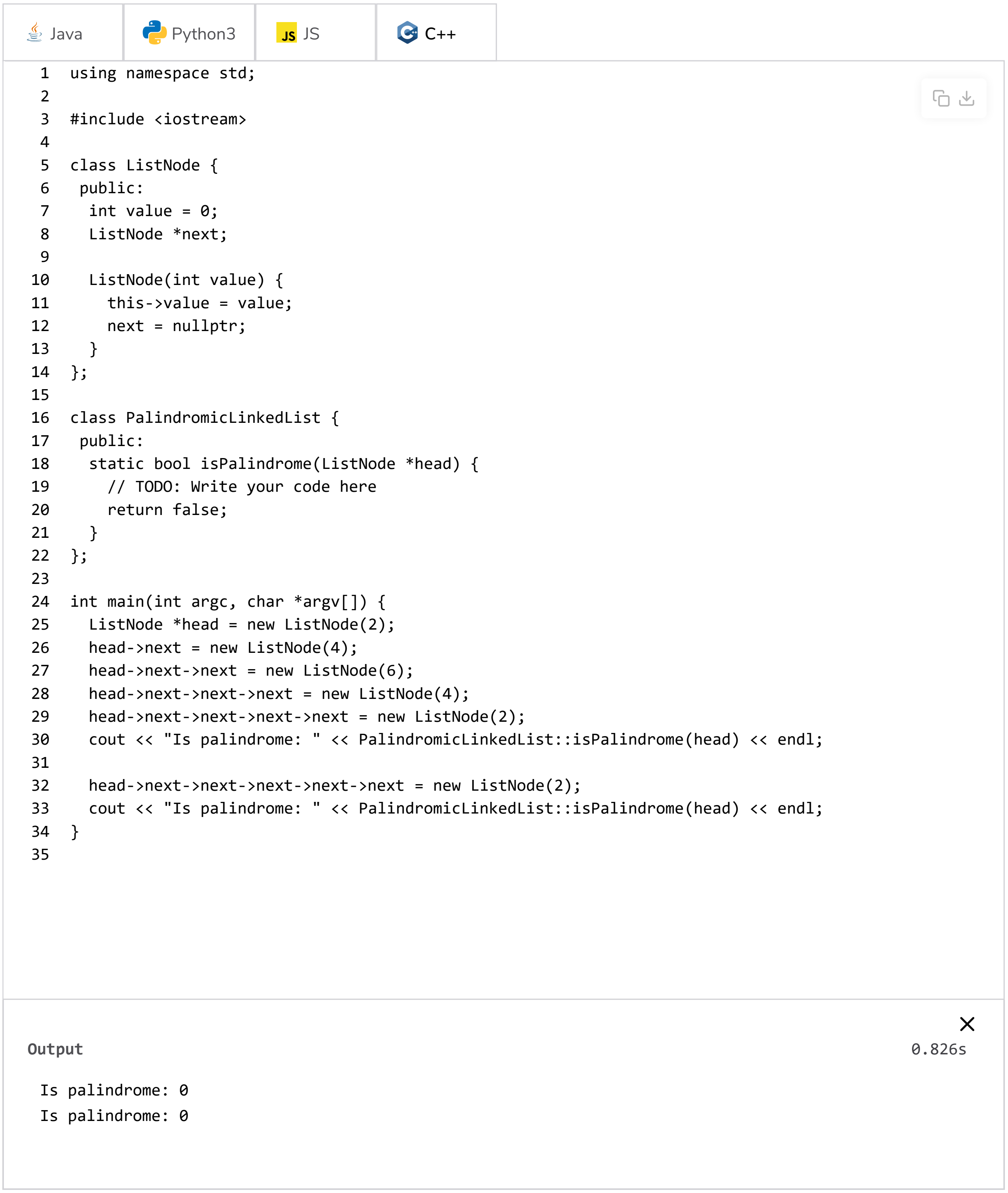
```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> null
Output: true
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> 2 -> null
Output: false
```

Try it yourself

Try solving this question here:



```cpp
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class PalindromicLinkedList {
 public:
  static bool isPalindrome(ListNode *head) {
    // TODO: Write your code here
    return false;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(2);
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(2);
  cout << "Is palindrome: " << PalindromicLinkedList::isPalindrome(head) << endl;

  head->next->next->next->next->next = new ListNode(2);
  cout << "Is palindrome: " << PalindromicLinkedList::isPalindrome(head) << endl;
}
```

Output 0.826s

```
Is palindrome: 0
Is palindrome: 0
```

Solution Review: Problem Challenge 1

Palindrome LinkedList (medium)

Given the head of a **Singly LinkedList**, write a method to check if the **LinkedList is a palindrome** or not.

Your algorithm should use **constant space** and the input LinkedList should be in the original form once the algorithm is finished. The algorithm should have $O(N)$ time complexity where 'N' is the number of nodes in the LinkedList.

Example 1:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> null
Output: true
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> 2 -> null
Output: false
```

Solution

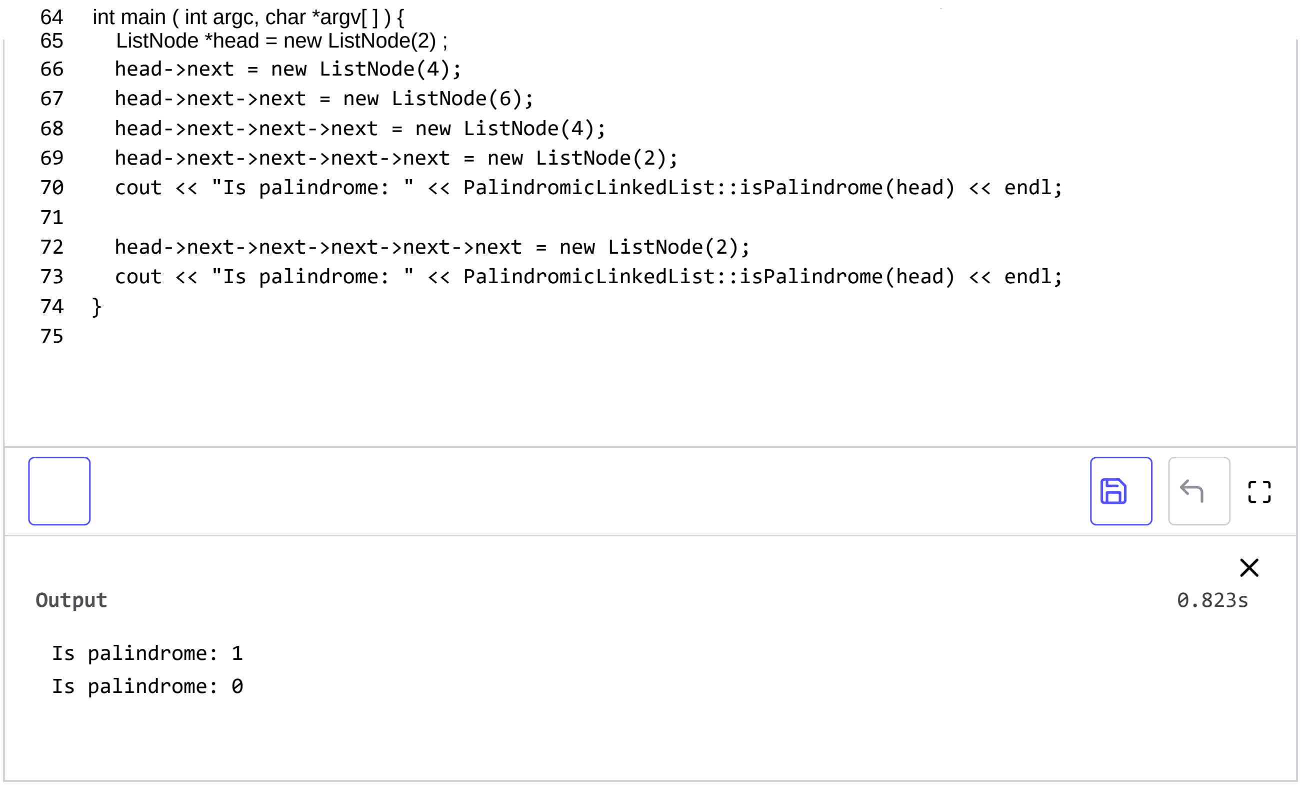
As we know, a palindrome LinkedList will have nodes values that read the same backward or forward. This means that if we divide the LinkedList into two halves, the node values of the first half in the forward direction should be similar to the node values of the second half in the backward direction. As we have been given a Singly LinkedList, we can't move in the backward direction. To handle this, we will perform the following steps:

1. We can use the **Fast & Slow pointers** method similar to Middle of the LinkedList to find the middle node of the LinkedList.
2. Once we have the middle of the LinkedList, we will reverse the second half.
3. Then, we will compare the first half with the reversed second half to see if the LinkedList represents a palindrome.
4. Finally, we will reverse the second half of the LinkedList again to revert and bring the LinkedList back to its original form.

Code

Java Python3 C++ JS

```cpp
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class PalindromicLinkedList {
 public:
  static bool isPalindrome(ListNode *head) {
    if (head == nullptr || head->next == nullptr) {
      return true;
    }

    // find middle of the LinkedList
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
    }

    ListNode *headSecondHalf = reverse(slow);  // reverse the second half
    ListNode *copyHeadSecondHalf =
        headSecondHalf;  // store the head of reversed part to revert back later

    // compare the first and the second half
    while (head != nullptr && headSecondHalf != nullptr) {
      if (head->value != headSecondHalf->value) {
        break;  // not a palindrome
      }
      head = head->next;
      headSecondHalf = headSecondHalf->next;
    }

    reverse(copyHeadSecondHalf);                         // revert the reverse of the second half
    if (head == nullptr || headSecondHalf == nullptr) {  // if both halves match
      return true;
    }
    return false;
  }

 private:
  static ListNode *reverse(ListNode *head) {
    ListNode *prev = nullptr;
    while (head != nullptr) {
      ListNode *next = head->next;
      head->next = prev;
      prev = head;
      head = next;
    }
    return prev;
  }
};

```



```
int main ( int argc, char *argv[ ] ) {
  ListNode *head = new ListNode(2) ;
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(4);
  head->next->next->next->next = new ListNode(2);
  cout << "Is palindrome: " << PalindromicLinkedList::isPalindrome(head) << endl;

  head->next->next->next->next->next = new ListNode(2);
  cout << "Is palindrome: " << PalindromicLinkedList::isPalindrome(head) << endl;
}
```

```
Is palindrome: 1
Is palindrome: 0
```

Time complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space complexity

The algorithm runs in constant space $O(1)$.

Problem Challenge 2

Rearrange a LinkedList (medium)

Given the head of a Singly LinkedList, write a method to modify the LinkedList such that the **nodes from the second half of the LinkedList are inserted alternately to the nodes from the first half in reverse order**. So if the LinkedList has nodes 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null, your method should return 1 -> 6 -> 2 -> 5 -> 3 -> 4 -> null.

Your algorithm should not use any extra space and the input LinkedList should be modified in-place.

Example 1:

```
Input: 2 -> 4 -> 6 -> 8 -> 10 -> 12 -> null
Output: 2 -> 12 -> 4 -> 10 -> 6 -> 8 -> null
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 8 -> 10 -> null
Output: 2 -> 10 -> 4 -> 8 -> 6 -> null
```

Try it yourself

Try solving this question here:

Java Python3 JS C++

```
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class RearrangeList {
 public:
  static void reorder(ListNode *head) {
    // TODO: Write your code here
  }
};
```

```cpp

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(2);
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(8);
  head->next->next->next->next = new ListNode(10);
  head->next->next->next->next->next = new ListNode(12);
  RearrangeList::reorder(head);
  while (head != nullptr) {
    cout << head->value << " ";
    head = head->next;
  }
}

```

Output 0.874s

```
2 4 6 8 10 12
```

Solution Review: Problem Challenge 2

Rearrange a LinkedList (medium)

Given the head of a Singly LinkedList, write a method to modify the LinkedList such that the **nodes from the second half of the LinkedList are inserted alternately to the nodes from the first half in reverse order**. So if the LinkedList has nodes 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null, your method should return 1 -> 6 -> 2 -> 5 -> 3 -> 4 -> null.

Your algorithm should not use any extra space and the input LinkedList should be modified in-place.

Example 1:

```
Input: 2 -> 4 -> 6 -> 8 -> 10 -> 12 -> null
Output: 2 -> 12 -> 4 -> 10 -> 6 -> 8 -> null
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 8 -> 10 -> null
Output: 2 -> 10 -> 4 -> 8 -> 6 -> null
```

Solution

This problem shares similarities with Palindrome LinkedList. To rearrange the given LinkedList we will follow the following steps:

1. We can use the **Fast & Slow pointers** method similar to Middle of the LinkedList to find the middle node of the LinkedList.
2. Once we have the middle of the LinkedList, we will reverse the second half of the LinkedList.
3. Finally, we'll iterate through the first half and the reversed second half to produce a LinkedList in the required order.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```cpp
using namespace std;

#include <iostream>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    next = nullptr;
  }
};

class RearrangeList {
 public:
  static void reorder(ListNode *head) {
    if (head == nullptr || head->next == nullptr) {
      return;
    }

    // find the middle of the LinkedList
    ListNode *slow = head, *fast = head;
    while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
    }

    // slow is now pointing to the middle node
    ListNode *headSecondHalf = reverse(slow);  // reverse the second half
    ListNode *headFirstHalf = head;

    // rearrange to produce the LinkedList in the required order
    while (headFirstHalf != nullptr && headSecondHalf != nullptr) {
      ListNode *temp = headFirstHalf->next;
      headFirstHalf->next = headSecondHalf;
      headFirstHalf = temp;

      temp = headSecondHalf->next;
      headSecondHalf->next = headFirstHalf;
      headSecondHalf = temp;
    }

    // set the next of the last node to 'null'
    if (headFirstHalf != nullptr) {
      headFirstHalf->next = nullptr;
    }
  }

 private:
  static ListNode *reverse(ListNode *head) {
    ListNode *prev = nullptr;
    while (head != nullptr) {
      ListNode *next = head->next;
      head->next = prev;
      prev = head;
      head = next;
    }
    return prev;
  }
};

int main(int argc, char *argv[]) {
  ListNode *head = new ListNode(2);
  head->next = new ListNode(4);
  head->next->next = new ListNode(6);
  head->next->next->next = new ListNode(8);
  head->next->next->next->next = new ListNode(10);
  head->next->next->next->next->next = new ListNode(12);
  RearrangeList::reorder(head);
  while (head != nullptr) {
    cout << head->value << " ";
    head = head->next;
  }
}
```

```
Output                                                                    0.955s

 2 12 4 10 6 8
```

Time Complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space Complexity

The algorithm runs in constant space $O(1)$.

Problem Challenge 3

Cycle in a Circular Array (hard)

We are given an array containing positive and negative numbers. Suppose the array contains a number 'M' at a particular index. Now, if 'M' is positive we will move forward 'M' indices and if 'M' is negative move backwards 'M' indices. You should assume that the **array is circular** which means two things:

1. If, while moving forward, we reach the end of the array, we will jump to the first element to continue the movement.
2. If, while moving backward, we reach the beginning of the array, we will jump to the last element to continue the movement.

Write a method to determine **if the array has a cycle**. The cycle should have more than one element and should follow one direction which means the cycle should not contain both forward and backward movements.

Example 1:

```
Input: [1, 2, -1, 2, 2]
Output: true
Explanation: The array has a cycle among indices: 0 -> 1 -> 3 -> 0
```

Example 2:

```
Input: [2, 2, -1, 2]
Output: true
Explanation: The array has a cycle among indices: 1 -> 3 -> 1
```

Example 3:

```
Input: [2, 1, -1, -2]
Output: false
Explanation: The array does not have any cycle.
```

Try it yourself

Try solving this question here:

Java Python3 JS C++

```cpp
using namespace std;

#include <iostream>
#include <vector>

class CircularArrayLoop {
 public:
  static bool loopExists(const vector<int> &arr) {
    // TODO: Write your code here
    return false;
  }
};

int main(int argc, char *argv[]) {
  cout << CircularArrayLoop::loopExists(vector<int>{1, 2, -1, 2, 2}) << endl;
  cout << CircularArrayLoop::loopExists(vector<int>{2, 2, -1, 2}) << endl;
  cout << CircularArrayLoop::loopExists(vector<int>{2, 1, -1, -2}) << endl;
}
```

Output 0.901s

```
0
0
0
```

Solution Review: Problem Challenge 3

Cycle in a Circular Array (hard)

We are given an array containing positive and negative numbers. Suppose the array contains a number ‘M’ at a particular index. Now, if ‘M’ is positive we will move forward ‘M’ indices and if ‘M’ is negative move backwards ‘M’ indices. You should assume that the **array is circular** which means two things:

1. If, while moving forward, we reach the end of the array, we will jump to the first element to continue the movement.
2. If, while moving backward, we reach the beginning of the array, we will jump to the last element to continue the movement.

Write a method to determine **if the array has a cycle**. The cycle should have more than one element and should follow one direction which means the cycle should not contain both forward and backward movements.

Example 1:

```
Input: [1, 2, -1, 2, 2]
Output: true
Explanation: The array has a cycle among indices: 0 -> 1 -> 3 -> 0
```

Example 2:

```
Input: [2, 2, -1, 2]
Output: true
Explanation: The array has a cycle among indices: 1 -> 3 -> 1
```

Example 3:

```
Input: [2, 1, -1, -2]
Output: false
Explanation: The array does not have any cycle.
```

Solution

This problem involves finding a cycle in the array and, as we know, the **Fast & Slow pointer** method is an efficient way to do that. We can start from each index of the array to find the cycle.

If a number does not have a cycle we will move forward to the next element. There are a couple of additional things we need to take care of:

1. As mentioned in the problem, the cycle should have more than one element. This means that when we move a pointer forward, if the pointer points to the same element after the move, we have a one-element cycle. Therefore, we can finish our cycle search for the current element.

2. The other requirement mentioned in the problem is that the cycle should not contain both forward and backward movements. We will handle this by remembering the direction of each element while searching for the cycle. If the number is positive, the direction will be forward and if the number is negative, the direction will be backward. So whenever we move a pointer forward, if there is a change in the direction, we will finish our cycle search right there for the current element.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```
using namespace std;

#include <iostream>
#include <vector>

class CircularArrayLoop {
 public:
  static bool loopExists(const vector<int> &arr) {
    for (int i = 0; i < arr.size(); i++) {
      bool isForward = arr[i] >= 0;  // if we are moving forward or not
      int slow = i, fast = i;

      // if slow or fast becomes '-1' this means we can't find cycle for this number
      do {
        slow = findNextIndex(arr, isForward, slow);  // move one step for slow pointer
        fast = findNextIndex(arr, isForward, fast);  // move one step for fast pointer
        if (fast != -1) {
          fast = findNextIndex(arr, isForward, fast);  // move another step for fast pointer
        }
      } while (slow != -1 && fast != -1 && slow != fast);

      if (slow != -1 && slow == fast) {
        return true;
      }
    }

    return false;
  }

 private:
  static int findNextIndex(const vector<int> &arr, bool isForward, int currentIndex) {
    bool direction = arr[currentIndex] >= 0;
    if (isForward != direction) {
      return -1;  // change in direction, return -1
    }

    // wrap around for negative numbers
    int nextIndex = (currentIndex + arr[currentIndex] + arr.size()) % arr.size();

    // one element cycle, return -1
    if (nextIndex == currentIndex) {
      nextIndex = -1;
    }

    return nextIndex;
  }
};

int main(int argc, char *argv[]) {
  cout << CircularArrayLoop::loopExists(vector<int>{1, 2, -1, 2, 2}) << endl;
  cout << CircularArrayLoop::loopExists(vector<int>{2, 2, -1, 2}) << endl;
  cout << CircularArrayLoop::loopExists(vector<int>{2, 1, -1, -2}) << endl;
}
```

```
Output                                                          0.868s

 1
 1
 0
```

Time Complexity

The above algorithm will have a time complexity of $O(N^2)$ where 'N' is the number of elements in the array. This complexity is due to the fact that we are iterating all elements of the array and trying to find a cycle for each element.

Space Complexity

The algorithm runs in constant space $O(1)$.

An Alternate Approach

In our algorithm, we don't keep a record of all the numbers that have been evaluated for cycles. We know that all such numbers will not produce a cycle for any other instance as well. If we can remember all the numbers that have been visited, our algorithm will improve to $O(N)$ as, then, each number will be evaluated for cycles only once. We can keep track of this by creating a separate array however the space complexity of our algorithm will increase to $O(N)$.