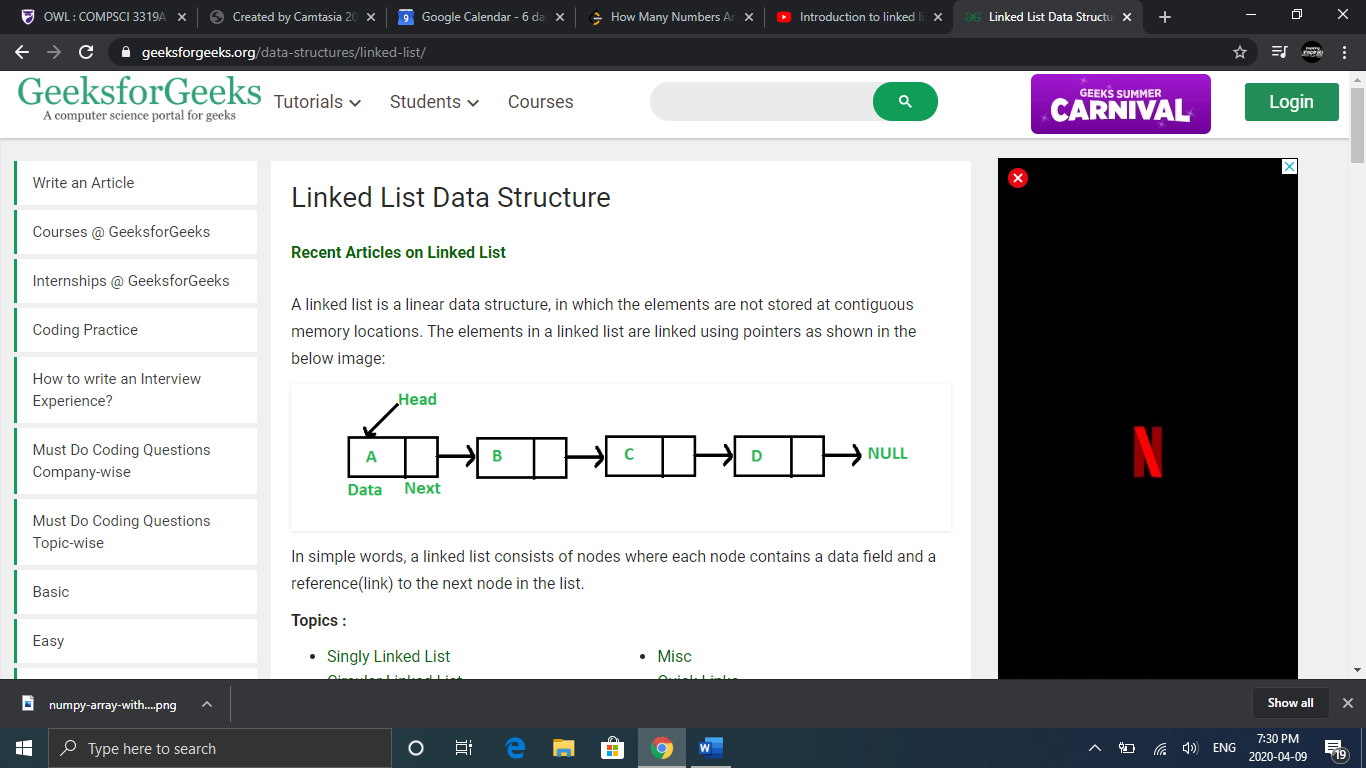
Linked Lists

* A linear data structure of elements which are stored at disjoint non-contiguous blocks of memory
* The elements in a linked list are linked using pointers



* As seen above, a block of memory in a Linked List is divided into two parts, the data or the value and the address of the next block
* These blocks of memory are also referred to as nodes
* The first node in a linked list is referred to as the head, if the linked list is empty, the value of the head node is NULL
* The only way to access all of the elements in a Linked List is by traversing from the head of the list to the end of the list. You know you have reached the end of the list when the next value pointed to is null

# Advantages

* Linked Lists are not of a fixed size (they are dynamic), so we can easily increase and decrease the size of these lists
* It is easy to insert and delete elements

# Disadvantages

* We have to access elements sequentially (no random access)
* Extra memory space is required for a pointer with each element of the list
* Not good for cache

# Java Implementation

class LinkedList {

    Node head; // head of list

    /\* Linked list Node.  This inner class is made static so that

       main() can access it \*/

    static class Node {

        int data;

        Node next;

        Node(int d)

        {

            data = d;

            next = null;

        } // Constructor

    }

    /\* method to create a simple linked list with 3 nodes\*/

    public static void main(String[] args)

    {

        /\* Start with the empty list. \*/

        LinkedList llist = new LinkedList();

        llist.head = new Node(1);

        Node second = new Node(2);

        Node third = new Node(3);

        /\* Three nodes have been allocated dynamically.

          We have references to these three blocks as head,

          second and third

          llist.head        second              third

             |                |                  |

             |                |                  |

         +----+------+     +----+------+     +----+------+

         | 1  | null |     | 2  | null |     |  3 | null |

         +----+------+     +----+------+     +----+------+ \*/

        llist.head.next = second; // Link first node with the second node

        /\*  Now next of the first Node refers to the second.  So they

            both are linked.

         llist.head        second              third

            |                |                  |

            |                |                  |

        +----+------+     +----+------+     +----+------+

        | 1  |  o-------->| 2  | null |     |  3 | null |

        +----+------+     +----+------+     +----+------+ \*/

        second.next = third; // Link second node with the third node

        /\*  Now next of the second Node refers to third.  So all three

            nodes are linked.

         llist.head        second              third

            |                |                  |

            |                |                  |

        +----+------+     +----+------+     +----+------+

        | 1  |  o-------->| 2  |  o-------->|  3 | null |

        +----+------+     +----+------+     +----+------+ \*/

    }

}

# Insertion

Suppose we are given the following Linked List data structure:

ListNode{

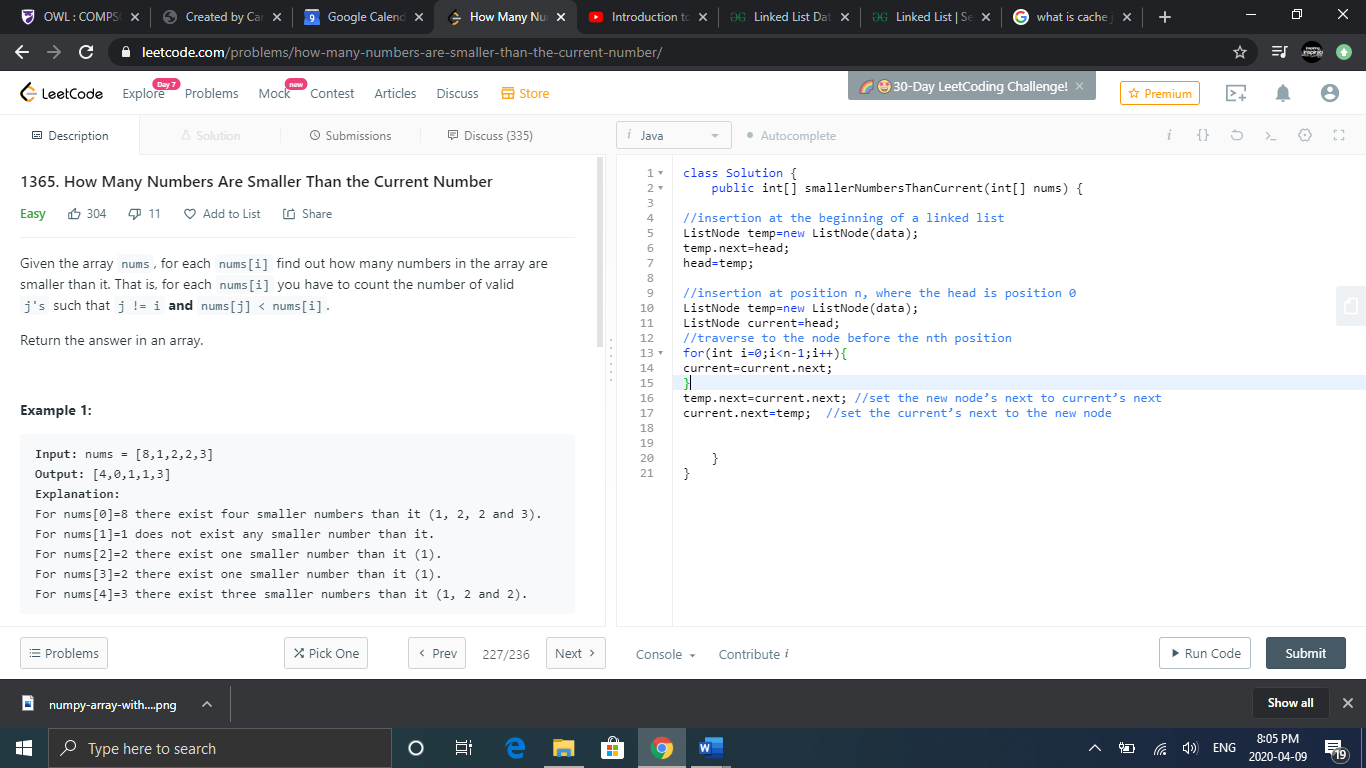
int data;

ListNode next;

ListNode(int val){ data=val;}

}

**Insertion at beginning of linked list**

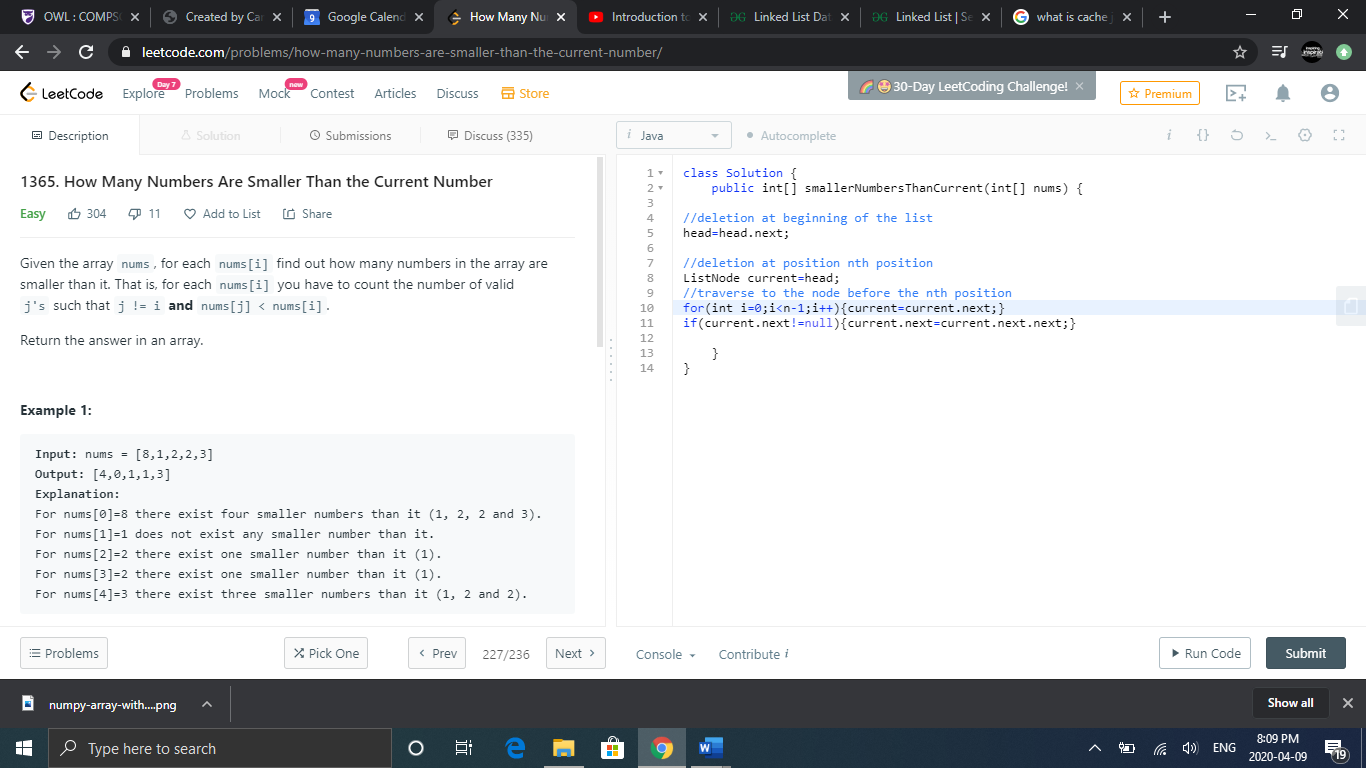


**Insertion at position n of linked list**

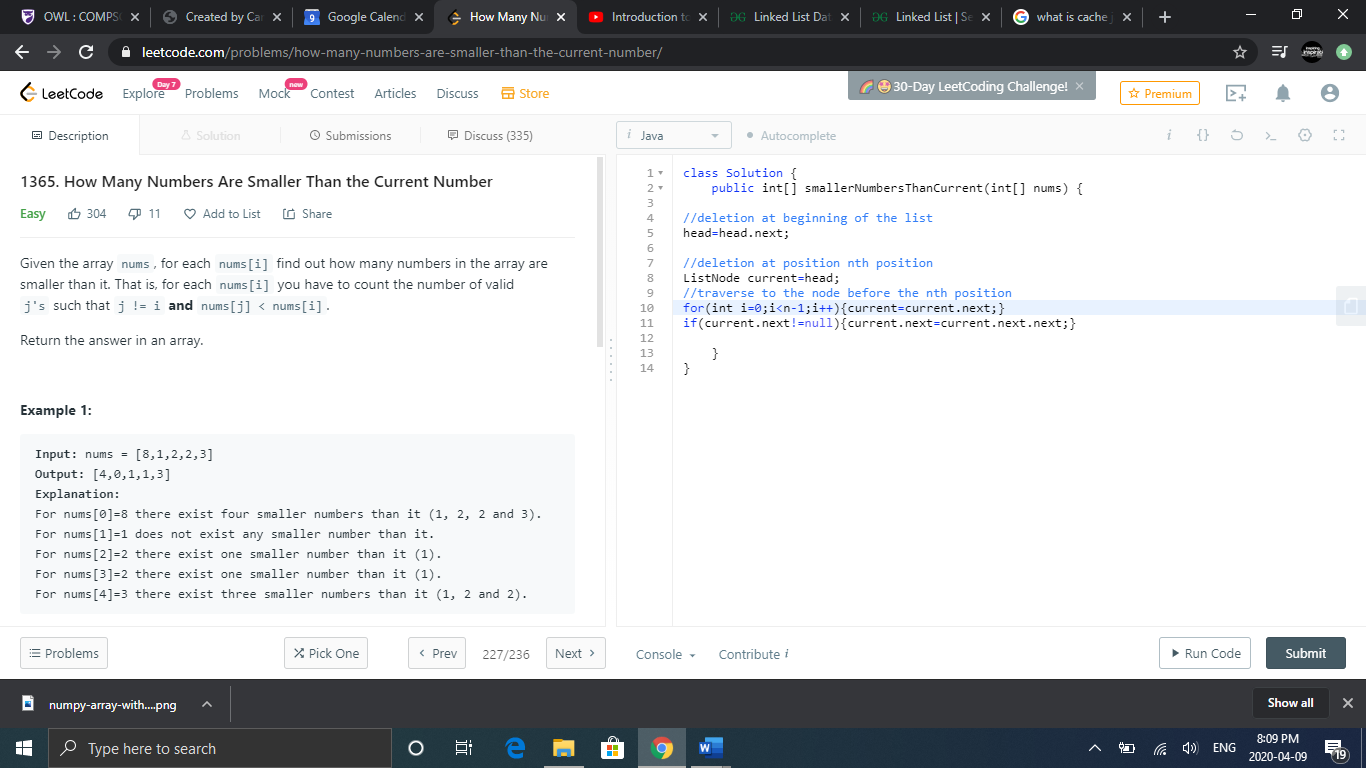
# 

# Deletion

**Deletion at beginning of linked list**

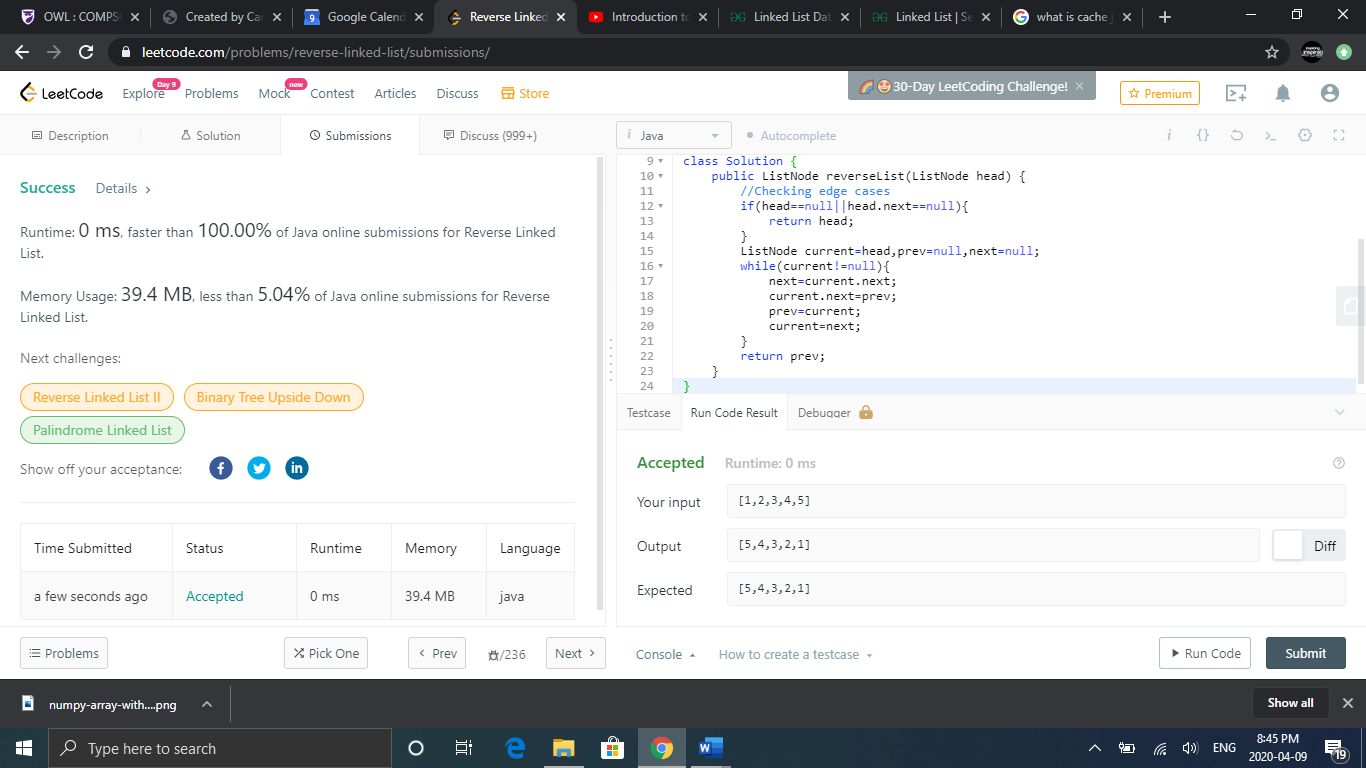


**Deletion at position n of linked list**



# Reverse a Linked List

**Iterative Implementation**

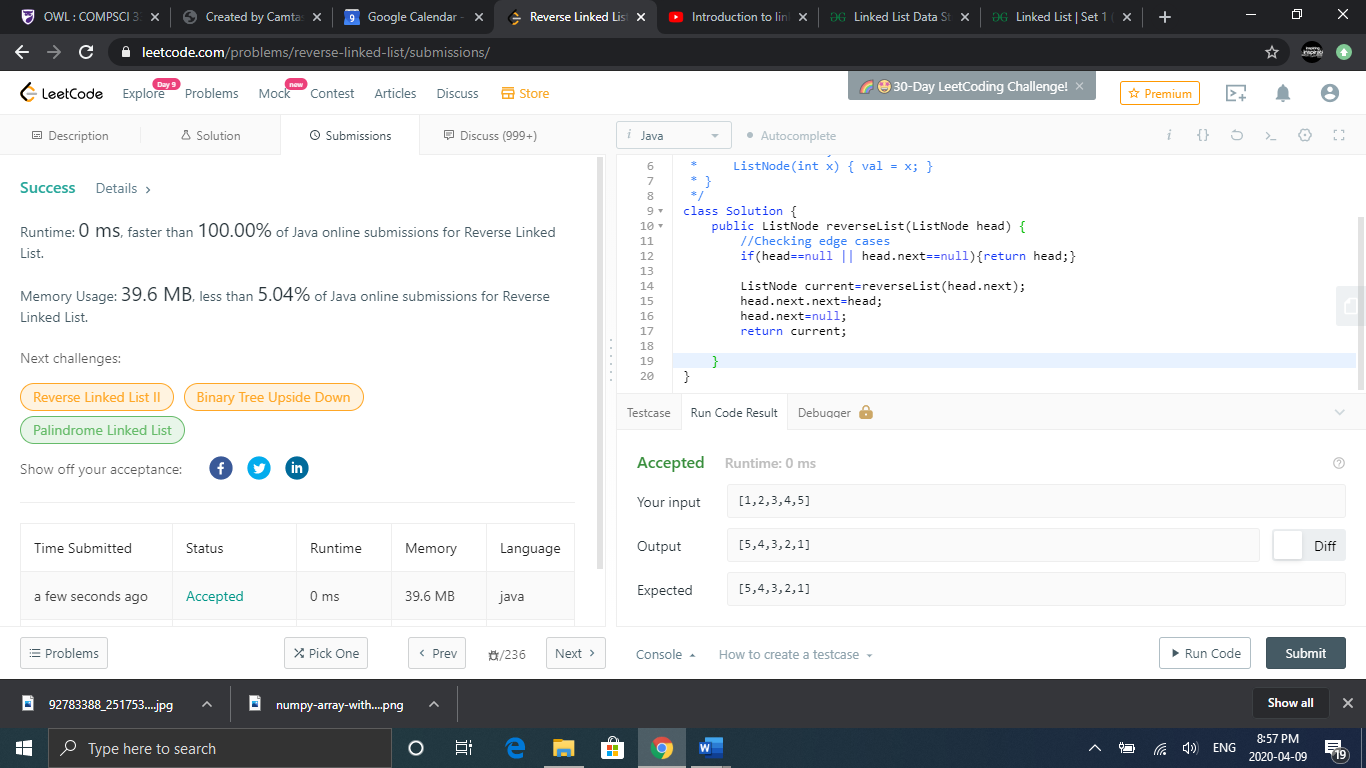


**Example:**

**A close up of text on a white background

Description automatically generated**

**Recursive Method**

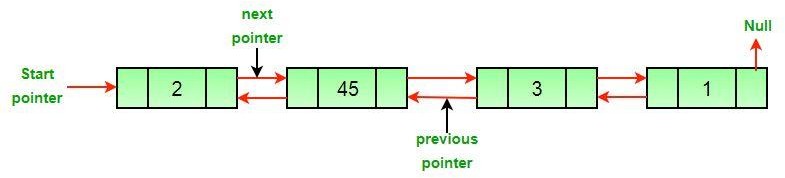


A close up of a whiteboard

Description automatically generated

# Doubly Linked List

* Same as Linked List except contains an extra pointer (previous pointer)



# Java Implementation

// Class for Doubly Linked List

public class DLL {

    Node head; // head of list

    /\* Doubly Linked list Node\*/

    class Node {

        int data;

        Node prev;

        Node next;

        // Constructor to create a new node

        // next and prev is by default initialized as null

        Node(int d) { data = d; }

    }

}

# Advantages

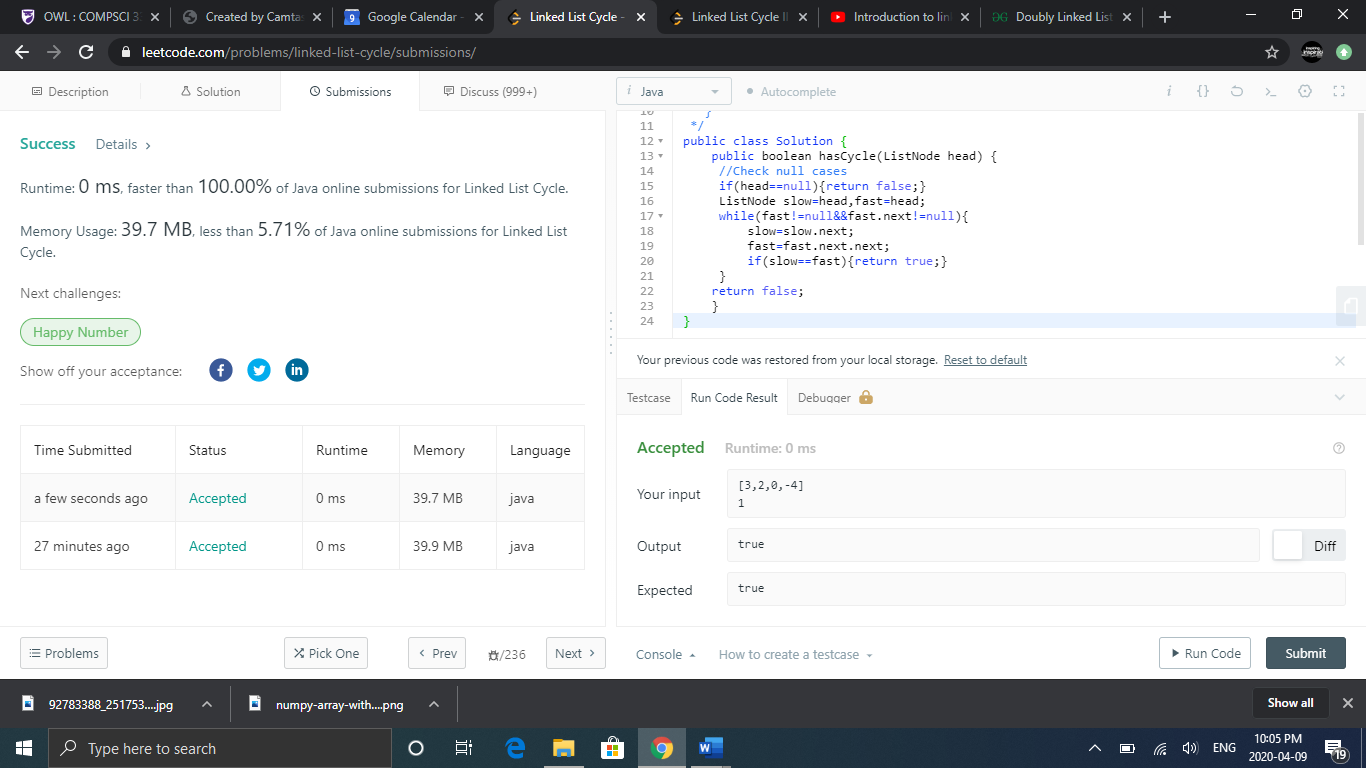
* Reverse traversal is possible
* delete operation in DLL is more efficient if pointer to the node to be deleted is given
* can quickly insert a new node before a given node

# Disadvantages

* Extra memory for pointer to previous node
* More links, more prone to errors when manipulating them

# Detect Cycle in Linked List

* Make two pointers: slow and fast
* Slow and fast start at head
* Make slow traverse one node at a time, make fast traverse 2 nodes at a time
* If they are ever equal, then there is a cycle



# Detect Start Of a Cycle

* Once you find that a cycle exists using the above algorithm, you can find the starting node of the cycle.
* Let x represent the distance from the start node to the start node of the cycle
* Let y represent the distance from the start of the cycle to where slow and fast meet
* Let z be the remaining distance from where slow and fast meet to the start of the cycle
* We will use the example: 1>2>3>4
* We know that fast covers twice the distance of slow so f=2s



* Our slow and fast would meet at the last node 4
* X=1>2, z=4>2, y= 2>3>4
* The distance fast covers is equal to x+y+z+y OR f=x+2y+z
* The distance slow covers is equal to x+y OR s=x+y
* We know f=2s, so we can say 2x+2y=x+2y+z
* Simplified that is x=z
* So, the distance from the start node to the start of the cycle is equal to the distance from where slow and fast meet to the start of the cycle
* So in our code, once slow and fast meet, we can move either our slow or fast pointer to the head and traverse the slow and fast pointers at the same speed until they meet again. The place they meet will be the start node of the cycle.

