Sum of the series

```
-----------------------------------------------
import java.io.*;
import java.util.*;

public class Solution {

    public static float sum_of_series(int i, int n, float s)
    {
        if(i>n)
            return s;
        else
        {
            if(i%2==0)
                s-=(float)1/i;
            else
                s+=(float)1/i;
            return sum_of_series(i+1,n,s);
        }
    }
    public static void main(String[] args) {
         Scanner scan=new Scanner(System.in);
        int n=scan.nextInt();

        double res=sum_of_series(1,n,0);
        System.out.println(res);
        /* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Soluti
on. */
    }
}

------------------------------------------------------------------------------------------------------------
```

Negative numbers

```
------------------------------------------
import java.io.*;
import java.util.*;

 class Solution {
     static void segregate(int arr[],
                    int n)
{

// Count negative numbers
int count_negative = 0;
for (int i = 0; i < n; i++)
    if (arr[i] < 0)
        count_negative++;

// Run a loop until all
// negative numbers are
// moved to the beginning
int i = 0, j = i + 1;
while (i != count_negative)
{

    // If number is negative,
    // update position of next
    // positive number.
    if (arr[i] < 0)
    {
        i++;
        j = i + 1;
    }

    // If number is positive, move
    // it to index j and increment j.
    else if (arr[i] > 0 && j < n)
    {
        int t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
        j++;
    }
}
}

    public static void main(String[] args)
    {
    int count_negative = 0;
    int arr[] = {-12, 11, -13, -5, 6, -7, 5, -3, -6 };
    int n = arr.length;
    segregate(arr, n);
    for (int i = 0; i < n; i++)
        System.out.print(arr[i] + " ");
}
}
```

Tower-of-Hanoi Problem

```
import java.io.*;
import java.util.*;

public class Solution {

  static void towerOfHanoi(int n, char from_rod,
                 char to_rod, char aux_rod)
{
   if (n == 0)
   {
      return;
   }
   towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
   System.out.println("Disk "+ n + " moved from " +
                  from_rod +" to " + to_rod );
   towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}
   public static void main(String[] args) {
      Scanner sc=new Scanner(System.in);
   int n = sc.nextInt();
   towerOfHanoi(n, 'A', 'C', 'B');
}
}
```

Max element

```
import java.io.*;
import java.util.*;

public class Solution {
    public static void maxelement(int no_of_rows, int[][] arr) {
      int i = 0;

      // Initialize max to 0 at beginning
      // of finding max element of each row
      int max = 0;
      int[] result = new int[no_of_rows];
      while (i < no_of_rows) {
         for (int j = 0; j < arr[i].length; j++) {
            if (arr[i][j] > max) {
               max = arr[i][j];
            }
         }
         result[i] = max;
         max =0;
         i++;

      }
      printArray(result);

   }

   // Print array element
   private static void printArray(int[] result) {
      for (int i =0; i<result.length;i++) {
         System.out.println(result[i]);
      }

   }
   public static void main(String[] args) {
      /* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Soluti
on. */
       int[][] arr = new int[][] { {1, 2, 3},{1, 4, 9},{76, 34, 21} };
     // Calling the function
      maxelement(3, arr);

   }
}
```

Pair with sum

```
import java.io.*;
import java.util.*;

public class Solution {

   public static void findPair(int[] A, int sum)
   {
      // consider each element except last element
      for (int i = 0; i < A.length - 1; i++)
      {
         // start from i'th element till last element
         for (int j = i + 1; j < A.length; j++)
         {
            // if desired sum is found, print it and return
            if (A[i] + A[j] == sum)
            {
               System.out.println("Pair found at index "
                           + i + " and " + j);
               return;
            }
         }
      }
      // No pair with given sum exists in the array
      System.out.println("Pair not found");
   }

   // main function
   public static void main (String[] args)
   {
      int A[] = { 5, 2, 6, 8, 1, 9 };
      int sum = 12;
      findPair(A, sum);
   }
}
```

==
========================

Find duplicates within a range k in an array. Given an array and a positive number k,
check whether the array contains any duplicate elements within the range k.If k is more than the array's size,
 the solution should check for duplicates in the complete array.

```
import java.io.*;
import java.util.*;

public class Solution {
   public static boolean hasDuplicate(int[] nums, int k)
   {
      // stores (element, index) pairs as (key, value) pairs
      Map<Integer, Integer> map = new HashMap<>();

      // traverse the array
      for (int i = 0; i < nums.length; i++)
      {
          // if the current element already exists in the map
          if (map.containsKey(nums[i]))
          {
             // return true if the current element repeats within range of `k`
             if (i - map.get(nums[i]) <= k) {
                return true;
             }
          }

          // store elements along with their indices
          map.put(nums[i], i);
       }

       // we reach here when no element repeats within range `k`
       return false;
    }


    public static void main(String[] args) {
       /* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Soluti
on. */
       int[] nums = { 5, 6, 8, 2, 4, 6, 9 };
       int k = 4;

       if (hasDuplicate(nums, k)) {
          System.out.println("Duplicates found");
       }
       else {
          System.out.println("No duplicates were found");
       }
    }

}
```

==
==

Find the smallest missing element from a sorted array Given a sorted array of distinct non-negative integers,
find the smallest missing element in it.

```
import java.io.*;
import java.util.*;

public class Solution {

    public static void main(String[] args) {

      Scanner sr = new Scanner(System.in);
       int n = sr.nextInt();
       int m = sr.nextInt();*/
       int arr[] ={0,1,2,6,9,11,15};

       int temp=0;
       for(int i=0;i<arr.length;i++)
```

```
        {
          if(arr[i]>i)
          {
            System.out.println("The smallest missing element is "+i);
            break;
          }
        }

    }
```
==

Bubble sort is a stable, in place srting algorithm named for smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even comparred to insertion sort, and is not recommended for large input.

```
import java.io.*;
import java.util.*;

public class Solution {
void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (arr[j] > arr[j + 1]) {

                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
    }


    public static void main(String[] args) {
         Solution ob = new Solution();
        int arr[] = {3, 5, 8, 4, 1, 9, -2 };
        ob.bubbleSort(arr);
        System.out.println(Arrays.toString(arr));

    }
}
```

==

Print the Elements of a Linked List

This is an to practice traversing a linked list. Given a pointer to the head node of a linked list, print each node's elem ent, one per line.
If the head pointer is null (indicating the list is empty), there is nothing to print.

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void insertNode(int nodeData) {
            SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

            if (this.head == null) {
                this.head = node;
            } else {
                this.tail.next = node;
            }

            this.tail = node;
        }
    }
 static void printLinkedList(SinglyLinkedListNode head) {
        SinglyLinkedListNode temp=head;
        while (temp!=null){
            System.out.println(temp.data);
            temp=temp.next;
        }

    }
 private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        SinglyLinkedList llist = new SinglyLinkedList();

        int llistCount = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");
        for (int i = 0; i < llistCount; i++) {
            int llistItem = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            llist.insertNode(llistItem);
        }

        printLinkedList(llist.head);

        scanner.close();
    }
}
```

==
==

Insert a node at the head of a linked list

Given a pointer to the head of a linked list, insert a new node before the head.
 The value in the new node should point to and the value should be replaced with a given value.
 Return a reference to the new head of the list. The head pointer given may be null meaning that the initial list is emp
ty.

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }


    }
    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter)
throws IOException {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));

            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }
 static SinglyLinkedListNode insertNodeAtHead(SinglyLinkedListNode llist, int data) {

         SinglyLinkedListNode temp=new SinglyLinkedListNode(data);
        if(llist == null){
            llist = new SinglyLinkedListNode(data);
            return llist;
        }
        temp.next = llist;
        llist = temp;
        return llist;


    }
 private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        SinglyLinkedList llist = new SinglyLinkedList();

        int llistCount = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int i = 0; i < llistCount; i++) {
            int llistItem = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

          SinglyLinkedListNode llist_head = insertNodeAtHead(llist.head, llistItem);

          llist.head = llist_head;
        }



        printSinglyLinkedList(llist.head, "\n", bufferedWriter);
        bufferedWriter.newLine();

        bufferedWriter.close();

        scanner.close();
    }
}
```

==
===

Insert a Node at the Tail of a Linked List

You are given the pointer to the head node of a linked list and an integer to add to the list. Create a new node with th
e given integer.
 Insert this node at the tail of the linked list and return the head node of the linked list formed after inserting this new
 node.
The given head pointer may be null, meaning that the initial list is empty.

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;

        public SinglyLinkedList() {
            this.head = null;
        }


    }

    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter)
throws IOException {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));

            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }
 static SinglyLinkedListNode insertNodeAtTail(SinglyLinkedListNode head, int data) {
        //SinglyLinkedListNode new_node=new SinglyLinkedListNode(data);
        if (head==null){
           head=new SinglyLinkedListNode(data);
           return head;
        }
        SinglyLinkedListNode temp=head;
          while(temp.next!=null){
             temp=temp.next;
          }
        temp.next=new SinglyLinkedListNode(data);

        return head;

    }

private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        SinglyLinkedList llist = new SinglyLinkedList();

        int llistCount = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int i = 0; i < llistCount; i++) {
            int llistItem = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

          SinglyLinkedListNode llist_head = insertNodeAtTail(llist.head, llistItem);

          llist.head = llist_head;
        }



        printSinglyLinkedList(llist.head, "\n", bufferedWriter);
        bufferedWriter.newLine();

        bufferedWriter.close();

        scanner.close();
    }
}
```

==
==

Insert a node at a specific position in a linked list

Given the pointer to the head node of a linked list and an integer to insert at a certain position, create a new node wit
h the given integer as its attribute,

insert this node at the desired position and return the head node.

```
public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void insertNode(int nodeData) {
            SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

            if (this.head == null) {
                this.head = node;
            } else {
                this.tail.next = node;
            }

            this.tail = node;
        }
    }

    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter)
throws IOException {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));

            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }

static SinglyLinkedListNode insertNodeAtPosition(SinglyLinkedListNode llist, int data, int position) {
     SinglyLinkedListNode node = new SinglyLinkedListNode(data);
        if(position==0){
            node.next = llist.next;
            llist=node;
        }else{
            SinglyLinkedListNode aux = new SinglyLinkedListNode(0);
            aux.next = llist;
            for(int i=0;i<position;i++)aux=aux.next;
            node.next = aux.next;
            aux.next=node;
        }

    return llist;


    }

private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        SinglyLinkedList llist = new SinglyLinkedList();

        int llistCount = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int i = 0; i < llistCount; i++) {
            int llistItem = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            llist.insertNode(llistItem);
        }

        int data = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        int position = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        SinglyLinkedListNode llist_head = insertNodeAtPosition(llist.head, data, position);

        printSinglyLinkedList(llist_head, " ", bufferedWriter);
        bufferedWriter.newLine();

        bufferedWriter.close();

        scanner.close();
    }
}
```
==

Delete a Node

Delete the node at a given position in a linked list and return a reference to the head node. The head is at position 0. The list may be empty after you delete the node.

In that case, return a null value.

```
class SinglyLinkedListNode {
    public int data;
    public SinglyLinkedListNode next;

    public SinglyLinkedListNode(int nodeData) {
        this.data = nodeData;
        this.next = null;
    }
}

class SinglyLinkedList {
    public SinglyLinkedListNode head;
    public SinglyLinkedListNode tail;

    public SinglyLinkedList() {
        this.head = null;
        this.tail = null;
    }

    public void insertNode(int nodeData) {
        SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

        if (this.head == null) {
            this.head = node;
        } else {
            this.tail.next = node;
        }

        this.tail = node;
    }
}

class SinglyLinkedListPrintHelper {
    public static void printList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter) throws IOEx
ception {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));

            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }
}


class Result{
static SinglyLinkedListNode deleteNode(SinglyLinkedListNode llist, int position) {
        int currentNodePosition = 0;
        SinglyLinkedListNode head = llist;
        SinglyLinkedListNode currentNode = llist;

         if (position == 0) {
            head = head.next;
            return head;
        }

        while (currentNodePosition < position - 1) {
            currentNode = currentNode.next;
            currentNodePosition++;
        }

        if (currentNode.next != null && currentNode.next.next != null) {
            currentNode.next = currentNode.next.next;
        }

        return head;
    }
}
public class Solution {
    public static void main(String[] args) throws IOException {
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        SinglyLinkedList llist = new SinglyLinkedList();

        int llistCount = Integer.parseInt(bufferedReader.readLine().trim());

        IntStream.range(0, llistCount).forEach(i -> {
            try {
                int llistItem = Integer.parseInt(bufferedReader.readLine().trim());

                llist.insertNode(llistItem);
            } catch (IOException ex) {
                throw new RuntimeException(ex);
            }
        });

        int position = Integer.parseInt(bufferedReader.readLine().trim());

        SinglyLinkedListNode llist1 = Result.deleteNode(llist.head, position);

        SinglyLinkedListPrintHelper.printList(llist1, " ", bufferedWriter);
        bufferedWriter.newLine();

        bufferedReader.close();
        bufferedWriter.close();
    }
}
```

==

==
Delete duplicate-value nodes from a sorted linked list

You are given the pointer to the head node of a sorted linked list, where the data in the nodes is in ascending order. Delete nodes and return a sorted list with each distinct value in the original list. The given head pointer may be null indicating that the list is empty.

```
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void insertNode(int nodeData) {
            SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

            if (this.head == null) {
                this.head = node;
            } else {
                this.tail.next = node;
            }

            this.tail = node;
        }
    }

    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter)
throws IOException {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));
            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }

public static SinglyLinkedListNode removeDuplicates(SinglyLinkedListNode llist) {

    SinglyLinkedListNode temp = llist;

        while(temp.next!=null)
        {
            if(temp.data == temp.next.data)
            {

                    temp.next = temp.next.next;
            }
               else
               {
                 temp = temp.next;
               }
        }
        return llist;
}


class Result{


}
private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        int t = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int tItr = 0; tItr < t; tItr++) {
            SinglyLinkedList llist = new SinglyLinkedList();

            int llistCount = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            for (int i = 0; i < llistCount; i++) {
                int llistItem = scanner.nextInt();
                scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

                llist.insertNode(llistItem);
            }

            SinglyLinkedListNode llist1 = removeDuplicates(llist.head);
            printSinglyLinkedList(llist1, " ", bufferedWriter);
            bufferedWriter.newLine();
        }

        bufferedWriter.close();

        scanner.close();
    }
}
```

==
========

Merge two sorted linked lists

Given pointers to the heads of two sorted linked lists, merge them into a single, sorted linked list.
Either head pointer may be null meaning that the corresponding list is empty.

```
public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void insertNode(int nodeData) {
            SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

            if (this.head == null) {
                this.head = node;
            } else {
                this.tail.next = node;
            }

            this.tail = node;
        }
    }

    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter)
```

```
throws IOException {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));

            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }
static SinglyLinkedListNode mergeLists(SinglyLinkedListNode head1, SinglyLinkedListNode head2) {
            if(head1==null) {
            return head2;
        }
        if(head2 == null) {
            return head1;
        }
       SinglyLinkedListNode t1 = head1, t2 = head2;
       SinglyLinkedListNode head = null, tail = null;
       if(t1.data<=t2.data) {
           head = t1;
           tail = t1;
           t1= t1.next;
       } else {
           head = t2;
           tail = t2;
           t2 =  t2.next;
       }
       while(t1!=null && t2!=null) {
           if(t1.data<=t2.data) {
               tail.next = t1;
               tail = t1;
               t1 = t1.next;
           } else {
               tail.next = t2;
               tail = t2;
               t2 = t2.next;
           }

       }
       if(t1!=null) {
           tail.next = t1;
       } else {
           tail.next = t2;
       }
       return head;



    }

 private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        int tests = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int testsItr = 0; testsItr < tests; testsItr++) {
            SinglyLinkedList llist1 = new SinglyLinkedList();

            int llist1Count = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            for (int i = 0; i < llist1Count; i++) {
                int llist1Item = scanner.nextInt();
                scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

                llist1.insertNode(llist1Item);
            }

           SinglyLinkedList llist2 = new SinglyLinkedList();

            int llist2Count = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            for (int i = 0; i < llist2Count; i++) {
                int llist2Item = scanner.nextInt();
                scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

                llist2.insertNode(llist2Item);
            }

            SinglyLinkedListNode llist3 = mergeLists(llist1.head, llist2.head);

            printSinglyLinkedList(llist3, " ", bufferedWriter);
            bufferedWriter.newLine();
        }

        bufferedWriter.close();

        scanner.close();
    }
}
```

==
===

Find Merge Point of Two Lists

Given pointers to the head nodes of linked lists that merge together at some point,
 find the node where the two lists merge. The merge point is where both lists point to the same node,
 i.e. they reference the same memory location. It is guaranteed that the two head nodes will be different,
 and neither will be NULL. If the lists share a common node, return that node's value.

```
public class Solution {
    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void insertNode(int nodeData) {
            SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

            if (this.head == null) {
                this.head = node;
            } else {
                this.tail.next = node;
            }

            this.tail = node;
        }
    }

    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep, BufferedWriter bufferedWriter)
throws IOException {
        while (node != null) {
            bufferedWriter.write(String.valueOf(node.data));

            node = node.next;

            if (node != null) {
                bufferedWriter.write(sep);
            }
        }
    }
  static int findMergeNode(SinglyLinkedListNode head1, SinglyLinkedListNode head2) {
        SinglyLinkedListNode temp1=head1;
    SinglyLinkedListNode temp2=head2;
    List<SinglyLinkedListNode>list=new ArrayList<SinglyLinkedListNode>();
    while(temp1!=null){
        list.add(temp1);
        temp1=temp1.next;
    }
    while(temp2!=null){
        if(list.contains(temp2)){
            break;
        }
        temp2=temp2.next;
    }
    return temp2.data;

    }

private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT_PATH")));

        int tests = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int testsItr = 0; testsItr < tests; testsItr++) {
            int index = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            SinglyLinkedList llist1 = new SinglyLinkedList();

            int llist1Count = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            for (int i = 0; i < llist1Count; i++) {
                int llist1Item = scanner.nextInt();
                scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

                llist1.insertNode(llist1Item);
            }

           SinglyLinkedList llist2 = new SinglyLinkedList();

            int llist2Count = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            for (int i = 0; i < llist2Count; i++) {
                int llist2Item = scanner.nextInt();
                scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

                llist2.insertNode(llist2Item);
            }

           SinglyLinkedListNode ptr1 = llist1.head;
            SinglyLinkedListNode ptr2 = llist2.head;

            for (int i = 0; i < llist1Count; i++) {
                if (i < index) {
                    ptr1 = ptr1.next;
                }
            }

            for (int i = 0; i < llist2Count; i++) {
                if (i != llist2Count-1) {
                    ptr2 = ptr2.next;
                }
            }

            ptr2.next = ptr1;

            int result = findMergeNode(llist1.head, llist2.head);

            bufferedWriter.write(String.valueOf(result));
            bufferedWriter.newLine();
        }

        bufferedWriter.close();

        scanner.close();
    }
}
```

===

Print in Reverse

Given a pointer to the head of a singly-linked list, print each value from the reversed list.
 If the given list is empty, do not print anything.

```
public class Solution {

    static class SinglyLinkedListNode {
        public int data;
        public SinglyLinkedListNode next;

        public SinglyLinkedListNode(int nodeData) {
            this.data = nodeData;
            this.next = null;
        }
    }

    static class SinglyLinkedList {
        public SinglyLinkedListNode head;
        public SinglyLinkedListNode tail;

        public SinglyLinkedList() {
            this.head = null;
            this.tail = null;
        }

        public void insertNode(int nodeData) {
            SinglyLinkedListNode node = new SinglyLinkedListNode(nodeData);

            if (this.head == null) {
                this.head = node;
            } else {
                this.tail.next = node;
            }

            this.tail = node;
        }
    }

    public static void printSinglyLinkedList(SinglyLinkedListNode node, String sep) {
        while (node != null) {
            System.out.print(node.data);

            node = node.next;

            if (node != null) {
                System.out.print(sep);
            }
        }
    }
    public static void reversePrint(SinglyLinkedListNode llist) {
    // Write your code here

         if(llist.next != null) {
            reversePrint(llist.next);
        }
        System.out.println(llist.data);


}
 private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        int tests = scanner.nextInt();
        scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

        for (int testsItr = 0; testsItr < tests; testsItr++) {
            SinglyLinkedList llist = new SinglyLinkedList();

            int llistCount = scanner.nextInt();
            scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

            for (int i = 0; i < llistCount; i++) {
                int llistItem = scanner.nextInt();
                scanner.skip("(\r\n|[\n\r\u2028\u2029\u0085])?");

                llist.insertNode(llistItem);
            }

            reversePrint(llist.head);
        }

        scanner.close();
    }
}
```

==
=======================
imp
queue using linked list

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;

public class Solution {
   QNode front, rear;

   static class QNode {
      int key;
      QNode next;

      // constructor to create a new linked list node
      public QNode(int key)
      {
         this.key = key;
         this.next = null;
      }
   }
   public Solution()
   {
      this.front = this.rear = null;
   }

   // Method to add an key to the queue.
   void Insert(int key)
   {
      QNode temp = new QNode(key);

          if (this.rear == null) {
         this.front = this.rear = temp;
         return;
      }
      this.rear.next = temp;
      this.rear = temp;
   }


   void Delete()
   {

      if (this.front == null)
         System.out.println("Queue is empty");

      QNode temp = this.front;
```

```
        if(temp!=null)
        this.front = this.front.next;
        else if (this.front == null)
            this.rear = null;
    }

    void Display()
    {
        if (this.front == null )
            System.out.print("NULL");
        QNode temp= front;
        while(temp!=null)
        {
        System.out.print("->"+temp.key);
        temp=temp.next;
        }
        System.out.println("");
    }

    void Exit(){

     System.exit(0);
    }

    public static void main(String[] args)
    {
        Solution S = new Solution();
        Scanner sc1=new Scanner(System.in);
        while(true)
        {
        int choice= sc1.nextInt();
        switch(choice)
        {
        case 1:
        int key=sc1.nextInt();
        S.Insert(key);
        break;

        case 2:
        S.Delete();
        break;

        case 3:
        S.Display();
        break;

        case 4:
        S.Exit();
        break;
        }
        }
}
}
```

==
==

Tree: Height of a Binary Tree

The height of a binary tree is the number of edges between the tree's root and its furthest leaf.
For example, the following binary tree is of height :

```
class Node {
    Node left;
    Node right;
    int data;

    Node(int data) {
        this.data = data;
        left = null;
        right = null;
    }
}

class Solution {

public static int height(Node root) {
       //int height(Node root){
       if (root == null){
           return -1;
       }
       else{
           return 1+Math.max(height(root.left), height(root.right));
    }
   }

public static Node insert(Node root, int data) {
        if(root == null) {
            return new Node(data);
        } else {
            Node cur;
            if(data <= root.data) {
                cur = insert(root.left, data);
                root.left = cur;
            } else {
                cur = insert(root.right, data);
                root.right = cur;
            }
            return root;
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int t = scan.nextInt();
        Node root = null;
        while(t-- > 0) {
            int data = scan.nextInt();
            root = insert(root, data);
        }
        scan.close();
        int height = height(root);
        System.out.println(height);
    }
}
```

===

Check whether a Binary Tree is BST (Binary Search Tree) or not

Given a binary tree check whether it is a binary
search tree or not.

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;

class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class Solution
{
    // Root of the Binary Tree
    Node root;

    // To keep tract of previous node in Inorder Traversal
    Node prev;

    boolean isBST()  {
        prev = null;
        return isBST(root);
    }

    /* Returns true if given search tree is binary
       search tree (efficient version) */
    boolean isBST(Node node)
    {
        // traverse the tree in inorder fashion and
        // keep a track of previous node
        if (node != null)
        {
            if (!isBST(node.left))
                return false;

            // allows only distinct values node
            if (prev != null && node.data <= prev.data )
                return false;
            prev = node;
            return isBST(node.right);
        }
        return true;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        Solution tree = new Solution();
        tree.root = new Node(10);
        tree.root.left = new Node(20);
        tree.root.right = new Node(30);
        tree.root.left.left = new Node(40);
        tree.root.left.right = new Node(50);

        if (tree.isBST())
            System.out.println("IS BST");
        else
            System.out.println("Not a BST");
    }
}
```

==

Find the Number of Nodes in a Binary Search Tree

This section discusses the recursive algorithm which counts the size or total number of nodes in a Binary Search Tre e.

Total No. of nodes=Total No. of nodes in left sub-tree + Total no. of node in right sub-tree + 1

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;

import java.util.*;

class Solution{
   static node root;
```

```
// Structure of a Tree node
 static class node {
   int data;
   node left;
   node right;

   node(int data)
   {
   this.data = data;
   this.left = null;
   this.right = null;

   }
}


// Function to get the left height of
// the binary tree
static int left_height(node node)
{
   int ht = 0;
   while (node!=null) {
      ht++;
      node = node.left;
   }

   // Return the left height obtained
   return ht;
}

// Function to get the right height
// of the binary tree
public void insert(int data) {
       //Create a new node
       node newnode = new node(data);

       //Check whether tree is empty
       if(root == null){
          root = newnode;
          return;
        }
       else {
          //current node point to root of the tree
          node current = root, parent = null;

          while(true) {
             //parent keep track of the parent node of current node.
             parent = current;

             //If data is less than current's data, node will be inserted to the left of tree
             if(data < current.data) {
                current = current.left;
                if(current == null) {
                   parent.left = newnode;
                   return;
```

```
                    }
                }
                //If data is greater than current's data, node will be inserted to the right of tree
                else {
                    current = current.right;
                    if(current == null) {
                        parent.right = newnode;
                        return;
                    }
                }
            }
        }
    }

static int right_height(node node)
{
    int ht = 0;
    while (node!=null) {
        ht++;
        node = node.right;
    }

    // Return the right height obtained
    return ht;
}

// Function to get the count of nodes
// in complete binary tree
static int Totalnodes(node root)
{

    // Base Case
    if (root == null)
        return 0;

    // Find the left height and the
    // right heights
    int lh = left_height(root);
    int rh = right_height(root);

    // If left and right heights are
    // equal return 2^height(1<<height) -1
    if (lh == rh)
        return (1 << lh) - 1;

    // Otherwise, recursive call
    return 1 + Totalnodes(root.left)
           + Totalnodes(root.right);
}

// Helper function to allocate a new node
// with the given data
// Driver Code
public static void main(String[] args)
{
   Solution s=new Solution();
   Scanner sc=new Scanner(System.in);
   int d=sc.nextInt();
    s.insert(d);
    s.insert(d);
    s.insert(d);
    s.insert(d);
    s.insert(d);
    System.out.print("Total No. of Nodes in the BST = "+ Totalnodes(root));

}
}
```

==