Linked List

**1) Reverse Linked List (Leetcode - 206)**

Given the head of a singly linked list, reverse the list, and return the reversed list.

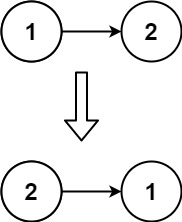
Example 1:



Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Example 3:

Input: head = []

Output: []

Constraints:

The number of nodes in the list is the range [0, 5000].

-5000 <= Node.val <= 5000

**Solution :**

class Solution {

    public ListNode reverseList(ListNode head) {

        ListNode next,prev,current;

        current=head;

        prev=null;

        while(current!=null)

        {

            next=current.next;

            current.next=prev;

            prev=current;

            current=next;

        }

        head=prev;

        return head;

    }

}

**2) Merge Two Sorted Lists (Leetcode - 21)**

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

The number of nodes in both lists is in the range [0, 50].

-100 <= Node.val <= 100

Both list1 and list2 are sorted in non-decreasing order.

**Solution :**

class Solution {

    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {

    if(list1!=null && list2!=null)

    {

        if(list1.val<list2.val)

        {

            list1.next=mergeTwoLists(list1.next,list2);

            return list1;

        }

        else if(list1.val>=list2.val)

        {

            list2.next=mergeTwoLists(list1,list2.next);

            return list2;

        }

    }

     if(list1==null)

    {

        return list2;

    }

    else

    {

        return list1;

    }

}

}

**3) Intersection of Two Linked Lists (Leetcode - 160)**

Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:

The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must retain their original structure after the function returns.

Custom Judge:

The inputs to the judge are given as follows (your program is not given these inputs):

intersectVal - The value of the node where the intersection occurs. This is 0 if there is no intersected node.

listA - The first linked list.

listB - The second linked list.

skipA - The number of nodes to skip ahead in listA (starting from the head) to get to the intersected node.

skipB - The number of nodes to skip ahead in listB (starting from the head) to get to the intersected node.

The judge will then create the linked structure based on these inputs and pass the two heads, headA and headB to your program. If you correctly return the intersected node, then your solution will be accepted.

Example 1:



Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

Output: Intersected at '8'

Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

- Note that the intersected node's value is not 1 because the nodes with value 1 in A and B (2nd node in A and 3rd node in B) are different node references. In other words, they point to two different locations in memory, while the nodes with value 8 in A and B (3rd node in A and 4th node in B) point to the same location in memory.

Example 2:



Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

Output: Intersected at '2'

Explanation: The intersected node's value is 2 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.

Example 3:



Input: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

Output: No intersection

Explanation: From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA and skipB can be arbitrary values.

Explanation: The two lists do not intersect, so return null.

Constraints:

The number of nodes of listA is in the m.

The number of nodes of listB is in the n.

1 <= m, n <= 3 \* 104

1 <= Node.val <= 105

0 <= skipA < m

0 <= skipB < n

intersectVal is 0 if listA and listB do not intersect.

intersectVal == listA[skipA] == listB[skipB] if listA and listB intersect.

**Solution :**

public class Solution {

    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

        ListNode l1=headA,l2=headB;

        while(l1!=l2)

        {

            l1 = l1==null?headB:l1.next;

            l2 = l2==null?headA:l2.next;

        }

        return l1;

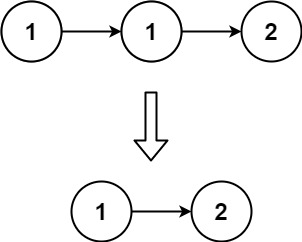
    }

}

**4) Remove Duplicates from Sorted List (Leetcode - -83)**

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

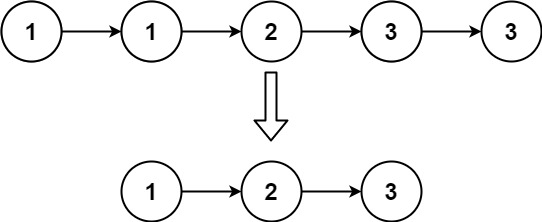
Example 1:



Input: head = [1,1,2]

Output: [1,2]

Example 2:



Input: head = [1,1,2,3,3]

Output: [1,2,3]

Constraints:

The number of nodes in the list is in the range [0, 300].

-100 <= Node.val <= 100

The list is guaranteed to be sorted in ascending order.

**Solution :**

class Solution {

    public ListNode deleteDuplicates(ListNode head) {

        ListNode current = head;

        while(current!=null)

        {

            while(current.next!=null && current.next.val==current.val)

            {

                current.next=current.next.next;

            }

            current=current.next;

        }

        return head;

    }

}

**5) Linked List Cycle (Leetcode - 141)**

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:



Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:



Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

Constraints:

The number of the nodes in the list is in the range [0, 104].

-105 <= Node.val <= 105

pos is -1 or a valid index in the linked-list.

**Solution :**

public class Solution {

    public boolean hasCycle(ListNode head) {

        ListNode first = head;

        ListNode second=head;

        boolean res = false;

        while(second!=null && second.next!=null)

        {

            first=first.next;

            second=second.next.next;

            if(first==second)

            {

                res=true;

                break;

            }

        }

        return res;

    }

}

**6) Palindrome Linked List (Leetcode - 234)**

Given the head of a singly linked list, return true if it is a

Palindrome or false otherwise.

Example 1:

Input: head = [1,2,2,1]

Output: true

Example 2:

Input: head = [1,2]

Output: false

Constraints:

The number of nodes in the list is in the range [1, 105].

0 <= Node.val <= 9

**Solution :**

**//space o(n) this is stack space**

class Solution {

    boolean ans=true;

    public ListNode reverse(ListNode head,ListNode temp)

    {

        if(head!=null && head.next==null)

        {

            if(temp.val!=head.val) ans=false;

            return temp.next;

        }

        ListNode r=reverse(head.next,temp);

        if(r.val!=head.val)

        {

            ans=false;

        }

        return r.next;

    }

    public boolean isPalindrome(ListNode head) {

        reverse(head,head);

        return ans;

    }

}

**//second solution space 0(1)**

class Solution {

    public ListNode reverse(ListNode head) {

        ListNode prev = null;

        ListNode curr = head;

        while(curr != null) {

            ListNode next = curr.next;

            curr.next = prev;

            prev = curr;

            curr = next;

        }

        return prev;

    }

    public boolean isPalindrome(ListNode head) {

        ListNode slow = head;

        ListNode fast = head.next;

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

            slow = slow.next;

            fast = fast.next.next;

        }

        ListNode rev = reverse(slow.next); // reverse second list

        slow.next = null;

        while(rev != null) {

            if(head.val != rev.val) {

                return false;

            }

            head = head.next;

            rev = rev.next;

        }

        return true;

    }

}

**1) Delete Node in a Linked List (Leetcode - 237) (Medium)**

There is a singly-linked list head and we want to delete a node node in it.

You are given the node to be deleted node. You will not be given access to the first node of head.

All the values of the linked list are unique, and it is guaranteed that the given node node is not the last node in the linked list.

Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

The value of the given node should not exist in the linked list.

The number of nodes in the linked list should decrease by one.

All the values before node should be in the same order.

All the values after node should be in the same order.

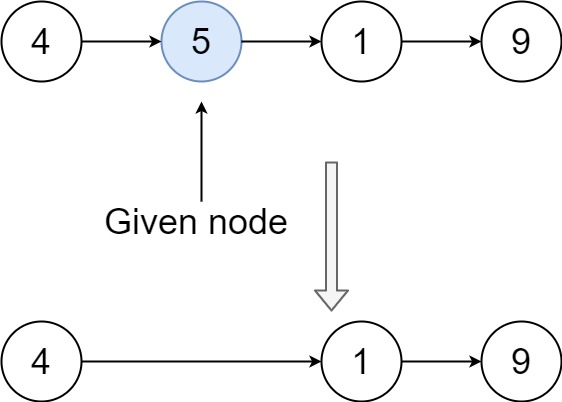
Custom testing:

For the input, you should provide the entire linked list head and the node to be given node. node should not be the last node of the list and should be an actual node in the list.

We will build the linked list and pass the node to your function.

The output will be the entire list after calling your function.

Example 1:

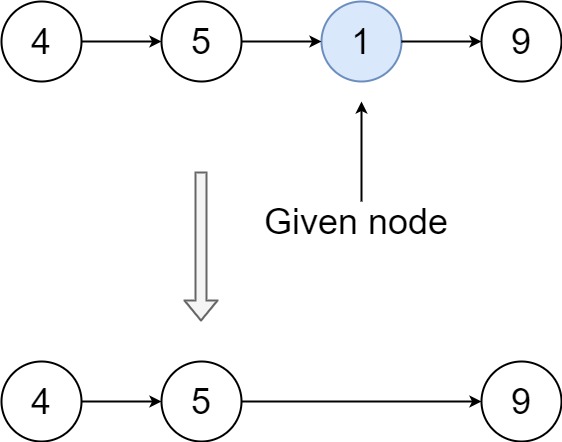


Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

Example 2:



Input: head = [4,5,1,9], node = 1

Output: [4,5,9]

Explanation: You are given the third node with value 1, the linked list should become 4 -> 5 -> 9 after calling your function.

Constraints:

The number of the nodes in the given list is in the range [2, 1000].

-1000 <= Node.val <= 1000

The value of each node in the list is unique.

The node to be deleted is in the list and is not a tail node.

**Solution :**

class Solution {

    public void deleteNode(ListNode node) {

        if(node!=null && node.next!=null)

        {

            node.val=node.next.val;

            node.next=node.next.next;

            node=node.next;

        }

    }

}

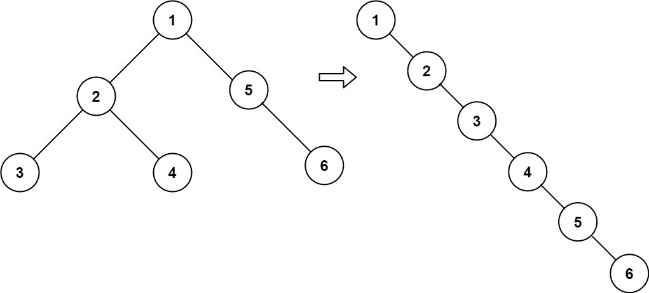
**2) Flatten Binary Tree to Linked List (Leetcode - 114)**

Given the root of a binary tree, flatten the tree into a "linked list":

The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.

The "linked list" should be in the same order as a pre-order traversal of the binary tree.

Example 1:



Input: root = [1,2,5,3,4,null,6]

Output: [1,null,2,null,3,null,4,null,5,null,6]

Example 2:

Input: root = []

Output: []

Example 3:

Input: root = [0]

Output: [0]

Constraints:

The number of nodes in the tree is in the range [0, 2000].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    public void flatten(TreeNode root) {

        if(root==null)

        {

          return ;

        }

        TreeNode temp = root;

        while(temp!=null)

        {

            if(temp.left!=null)

            {

                TreeNode current = temp.left;

                while(current.right!=null)

                {

                    current=current.right;

                }

                current.right=temp.right;

                temp.right=temp.left;

                temp.left=null;

            }

            temp=temp.right;

        }

    }

}

**3) Odd Even Linked List (Leetcode - 328)**

Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The first node is considered odd, and the second node is even, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in O(1) extra space complexity and O(n) time complexity.

Example 1:



Input: head = [1,2,3,4,5]

Output: [1,3,5,2,4]

Example 2:



Input: head = [2,1,3,5,6,4,7]

Output: [2,3,6,7,1,5,4]

Constraints:

The number of nodes in the linked list is in the range [0, 104].

-106 <= Node.val <= 106

**Solution :**

class Solution {

    public ListNode oddEvenList(ListNode head) {

     if(head == null || head.next == null) return head;

        ListNode odd = head;

        ListNode even = odd.next;

        ListNode evenHead = even;

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

            odd.next = even.next;

            odd = odd.next;

            even.next = odd.next;

            even = even.next;

        }

        odd.next = evenHead;

        return head;

    }

}

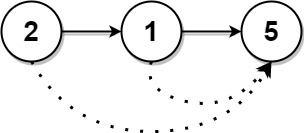
**4) Next Greater Node In Linked List (Leetcode - 1019)**

You are given the head of a linked list with n nodes.

For each node in the list, find the value of the next greater node. That is, for each node, find the value of the first node that is next to it and has a strictly larger value than it.

Return an integer array answer where answer[i] is the value of the next greater node of the ith node (1-indexed). If the ith node does not have a next greater node, set answer[i] = 0.

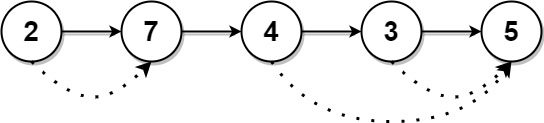
Example 1:



Input: head = [2,1,5]

Output: [5,5,0]

Example 2:



Input: head = [2,7,4,3,5]

Output: [7,0,5,5,0]

Constraints:

The number of nodes in the list is n.

1 <= n <= 104

1 <= Node.val <= 109

**Solution :**

class Solution {

    public int[] nextLargerNodes(ListNode head) {

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

        Stack <Integer> stack = new Stack<>();

        ListNode q1=head;

        while(q1!=null){

            list.add(q1.val);

            q1=q1.next;

        }

        int gse[] = new int[list.size()];

        for(int i=list.size()-1;i>=0;i--)

        {

            while(!stack.isEmpty() && list.get(i)>=list.get(stack.peek()))

            {

                stack.pop();

            }

            gse[i]=(!stack.isEmpty())?list.get(stack.peek()):0;

            stack.push(i);

        }

        return gse;

    }

}

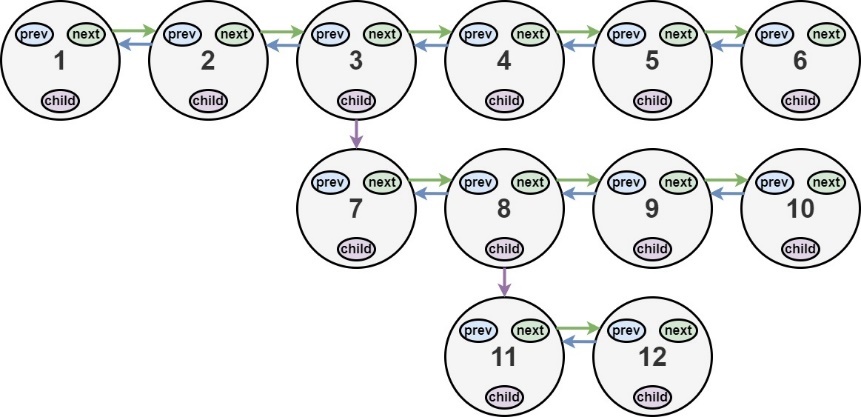
**5) Flatten a Multilevel Doubly Linked List (Leetcode - 430)**

You are given a doubly linked list, which contains nodes that have a next pointer, a previous pointer, and an additional child pointer. This child pointer may or may not point to a separate doubly linked list, also containing these special nodes. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure as shown in the example below.

Given the head of the first level of the list, flatten the list so that all the nodes appear in a single-level, doubly linked list. Let curr be a node with a child list. The nodes in the child list should appear after curr and before curr.next in the flattened list.

Return the head of the flattened list. The nodes in the list must have all of their child pointers set to null.

Example 1:

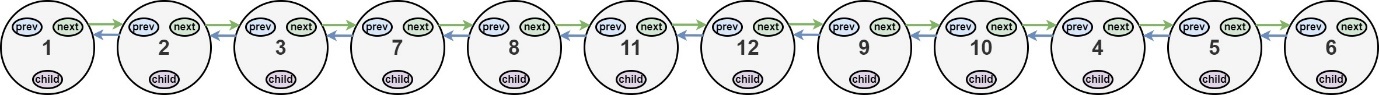


Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

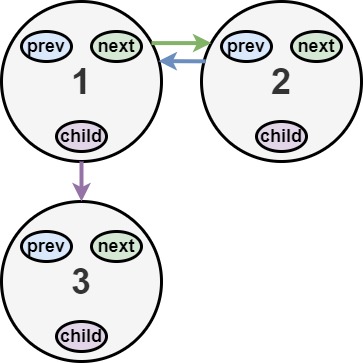
Output: [1,2,3,7,8,11,12,9,10,4,5,6]

Explanation: The multilevel linked list in the input is shown.

After flattening the multilevel linked list it becomes:



Example 2:



Input: head = [1,2,null,3]

Output: [1,3,2]

Explanation: The multilevel linked list in the input is shown.

After flattening the multilevel linked list it becomes:

Example 3:

Input: head = []

Output: []

Explanation: There could be empty list in the input.

Constraints:

The number of Nodes will not exceed 1000.

1 <= Node.val <= 105

**Solution :**

class Solution {

    public Node flatten(Node head) {

        Node curr = head;

        while(curr != null) {

            if(curr.child != null) {

                Node tail = helper(curr.child);

                if(curr.next != null)

                    curr.next.prev = tail;

                tail.next = curr.next;

                curr.next = curr.child;

                curr.child.prev = curr;

                curr.child = null;

            }

            curr = curr.next;

        }

        return head;

    }

    private Node helper(Node head) {

        while(head.next != null)

            head = head.next;

        return head;

    }

}

**6) Reorder List (Leetcode - 143)**

You are given the head of a singly linked-list. The list can be represented as:

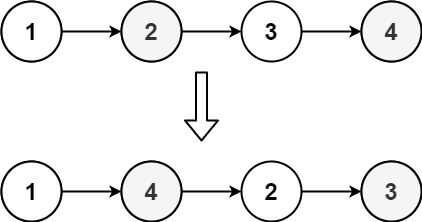
L0 → L1 → … → Ln - 1 → Ln

Reorder the list to be on the following form:

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

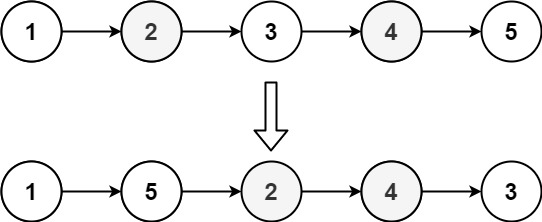
Example 1:



Input: head = [1,2,3,4]

Output: [1,4,2,3]

Example 2:



Input: head = [1,2,3,4,5]

Output: [1,5,2,4,3]

Constraints:

The number of nodes in the list is in the range [1, 5 \* 104].

1 <= Node.val <= 1000

**Solution :**

//first go to middle of list and reverse the second half

class Solution {

    public void reorderList(ListNode head) {

       ListNode fast = head;

       ListNode slow = head;

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

        {

            slow = slow.next;

            fast = fast.next.next;

        }

        ListNode prev = null;

        ListNode next = null;

        ListNode slow1 = slow;

        while(slow!=null)

        {

            next = slow.next;

            slow.next = prev;

            prev = slow;

            slow = next;

        }

        ListNode first=head;

        ListNode last=prev;

        while(last.next!=null)

        {

            next=first.next;

            prev=last.next;

            first.next=last;

            last.next=next;

            first=next;

            last=prev;

        }

    }

}

**7) Copy List with Random Pointer (Leetcode - 138)**

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a deep copy of the list. The deep copy should consist of exactly n brand new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return the head of the copied linked list.

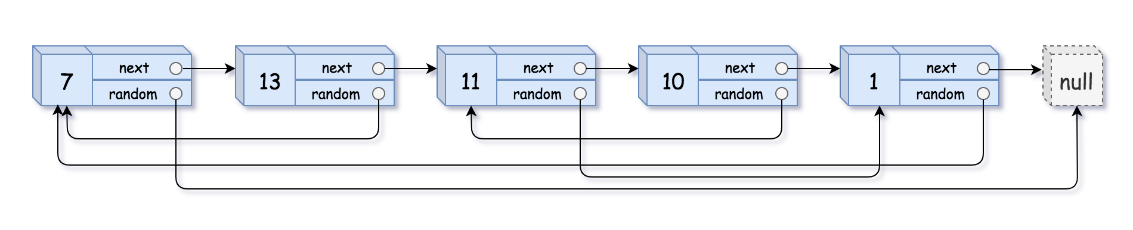
The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

val: an integer representing Node.val

random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will only be given the head of the original linked list.

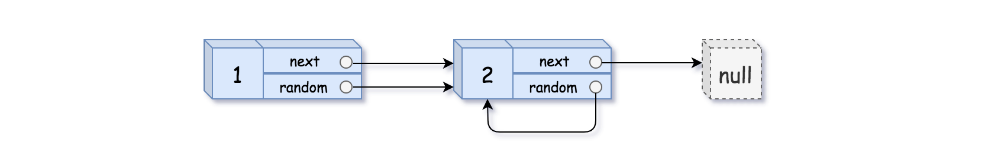
Example 1:



Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

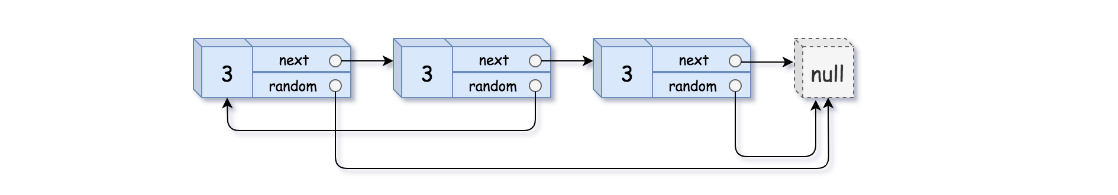
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]

Example 2:



Input: head = [[1,1],[2,1]]

Output: [[1,1],[2,1]]

Example 3:  


Input: head = [[3,null],[3,0],[3,null]]

Output: [[3,null],[3,0],[3,null]]

Constraints:

0 <= n <= 1000

-104 <= Node.val <= 104

Node.random is null or is pointing to some node in the linked list.

**Solution :**

class Solution {

    public Node copyRandomList(Node head) {

        Node current = head;

        Node temp=head;

        while(current!=null)

        {

            Node next = current.next;

            current.next=new Node(current.val);

            current.next.next=next;

            current=next;

        }

        while(temp!=null)

        {

           if(temp.random!=null)

           {

            temp.next.random=temp.random.next;

           }

           temp=temp.next.next;

        }

        current=head;

        Node dummy = new Node(-1);

        temp = dummy;

        while(current!=null && current.next!=null)

        {

            Node next=current.next;

            current.next=current.next.next;

            temp.next=next;

            temp=temp.next;

            current=current.next;

        }

        return dummy.next;

    }

}

**8) Linked List Cycle II (Leetcode - 142)**

Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

Example 1:



Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:



Input: head = [1,2], pos = 0

Output: tail connects to node index 0

Explanation: There is a cycle in the linked list, where tail connects to the first node.

Example 3:



Input: head = [1], pos = -1

Output: no cycle

Explanation: There is no cycle in the linked list.

Constraints:

The number of the nodes in the list is in the range [0, 104].

-105 <= Node.val <= 105

pos is -1 or a valid index in the linked-list.

**Solution :**

public class Solution {

    public ListNode detectCycle(ListNode head) {

        ListNode fast = head;

        ListNode slow = head;

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

        {

            fast=fast.next.next;

            slow=slow.next;

            if(fast==slow)

            {

                break;

            }

        }

        slow=head;

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

        {

            if(slow==fast)

            {

                return fast;

            }

            fast=fast.next;

            slow=slow.next;

        }

         return null;

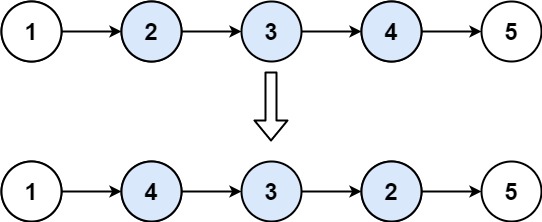
    }

}

**9) Reverse Linked List II (Leetcode - 92)**

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

Example 1:



Input: head = [1,2,3,4,5], left = 2, right = 4

Output: [1,4,3,2,5]

Example 2:

Input: head = [5], left = 1, right = 1

Output: [5]

Constraints:

The number of nodes in the list is n.

1 <= n <= 500

-500 <= Node.val <= 500

1 <= left <= right <= n

**Solution :**

class Solution {

    public ListNode reverseBetween(ListNode head, int left, int right) {

        ListNode temp=head;

        if(head == null || head.next == null) return head;

        ListNode prev=temp;

        for(int i=1;i<left;i++)

        {

            prev=temp;

            temp=temp.next;

        }

        ListNode temp1=temp;

        ListNode pre=null;

        ListNode next=null;

        for(int i=left;i<=right;i++)

        {

            next=temp1.next;

            temp1.next=pre;

            pre=temp1;

            temp1=next;

        }

        if(prev!=null)

        {

            prev.next=pre;

        }

        if(temp!=null)

{

temp.next=temp1;

}

        if(left==1)

        {

            return pre;

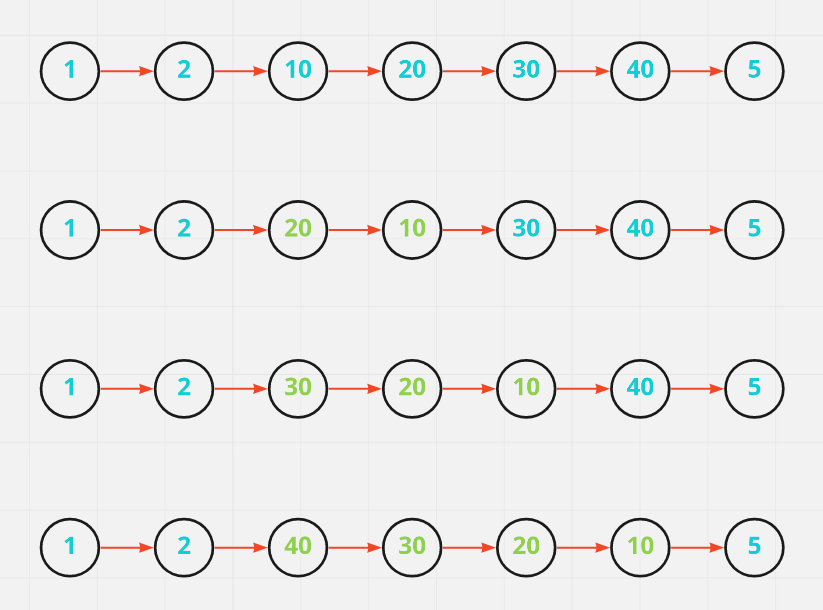
        }

    return head;

    }

}

**Solution :**



class Solution {

public ListNode reverseBetween(ListNode head, int left, int right) {

ListNode dummy = new ListNode(0); // created dummy node

dummy.next = head;

ListNode prev = dummy; // intialising prev pointer on dummy node

for(int i = 0; i < left - 1; i++)

prev = prev.next; // adjusting the prev pointer on it's actual index

ListNode curr = prev.next; // curr pointer will be just after prev

// reversing

for(int i = 0; i < right - left; i++){

ListNode forw = curr.next; // forw pointer will be after curr

curr.next = forw.next;

forw.next = prev.next;

prev.next = forw;

}

return dummy.next;

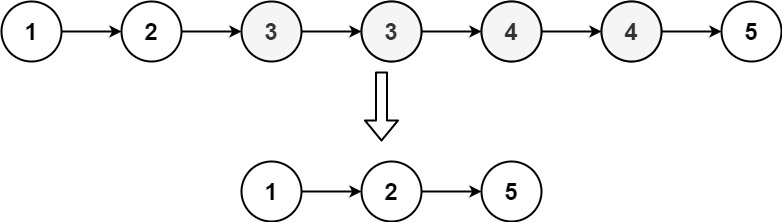
}

}

**10) Remove Duplicates from Sorted List II (Leetcode - 82)**

Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

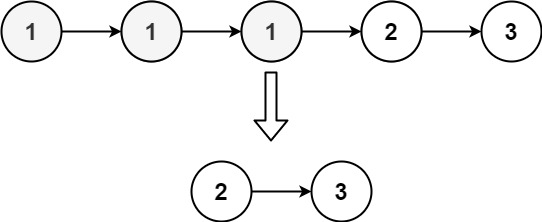
Example 1:



Input: head = [1,2,3,3,4,4,5]

Output: [1,2,5]

Example 2:



Input: head = [1,1,1,2,3]

Output: [2,3]

Constraints:

The number of nodes in the list is in the range [0, 300].

-100 <= Node.val <= 100

The list is guaranteed to be sorted in ascending order.

**Solution :**

class Solution {

    public ListNode deleteDuplicates(ListNode head) {

        ListNode dummy = new ListNode(-1);

        dummy.next=head;

        ListNode previous=dummy;

        ListNode curr = head;

        while(curr!=null)

        {

            while(curr.next!=null && curr.next.val==curr.val)

            {

                curr=curr.next;

            }

            if(previous.next==curr)

            {

                previous=previous.next;

            }

            else

            {

                previous.next=curr.next;

            }

            curr=curr.next;

        }

        return dummy.next;

    }

}

**11) Remove Nth Node From End of List (Leetcode - 19)**

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Example 1:



Input: head = [1,2,3,4,5], n = 2

Output: [1,2,3,5]

Example 2:

Input: head = [1], n = 1

Output: []

Example 3:

Input: head = [1,2], n = 1

Output: [1]

Constraints:

The number of nodes in the list is sz.

1 <= sz <= 30

0 <= Node.val <= 100

1 <= n <= sz

**Solution :**

class Solution {

    public ListNode removeNthFromEnd(ListNode head, int n)

    {

        int len=0;

        ListNode temp=head;

        while(temp!=null)

        {

            len+=1;

            temp=temp.next;

        }

        ListNode t=head;

        for(int i=1;i<len-n;i++)

        {

            t=t.next;

        }

        if(len==1)

        {

            head=null;

        }

        else if(len==n)

        {

            head=head.next;

        }

        else

        {

            t.next=t.next.next;

        }

        return head;

    }

}

**12) Add Two Numbers (Leetcode - 2)**

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:



Input: l1 = [2,4,3], l2 = [5,6,4]

Output: [7,0,8]

Explanation: 342 + 465 = 807.

Example 2:

Input: l1 = [0], l2 = [0]

Output: [0]

Example 3:

Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

Output: [8,9,9,9,0,0,0,1]

Constraints:

The number of nodes in each linked list is in the range [1, 100].

0 <= Node.val <= 9

It is guaranteed that the list represents a number that does not have leading zeros.

**Solution :**

class Solution {

    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

        ListNode dummy  = new ListNode(0);

        ListNode l3 = dummy;

        int carry = 0;

        int currSum = 0;

        while(l1!=null||l2!=null){

            int val1 = (l1 != null) ? l1.val : 0;

            int val2 = (l2 != null) ? l2.val : 0;

            currSum = val1+val2+carry;

            carry = currSum/10;

            int node = currSum%10;

            System.out.println(carry+" "+node);

            ListNode newNode = new ListNode(node);

            l3.next = newNode;

            if(l1!=null) l1 = l1.next;

            if(l2!=null) l2 = l2.next;

            l3 = l3.next;

        }

        System.out.println(carry);

        if(carry>0){

            ListNode newNode = new ListNode(carry);

            l3.next = newNode;

            l3 = l3.next;

        }

        return dummy.next;

    }

}

**13) LRU Cache (Leetcode - 146)**

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the LRUCache class:

LRUCache(int capacity) Initialize the LRU cache with positive size capacity.

int get(int key) Return the value of the key if the key exists, otherwise return -1.

void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.

The functions get and put must each run in O(1) average time complexity.

Example 1:

Input

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

Output

[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

Constraints:

1 <= capacity <= 3000

0 <= key <= 104

0 <= value <= 105

At most 2 \* 105 calls will be made to get and put.

**Solution :**

class LRUCache {

    private class Node

    {

        int key,val;

        Node next,prev;

        Node(int key,int val)

        {

            this.key=key;

            this.val=val;

        }

    }

    int capacity;

    Node left=new Node(0,0),right=new Node(0,0);

    HashMap<Integer,Node> cache;

    public LRUCache(int capacity) {

        this.capacity=capacity;

        cache = new HashMap<>();

        left.next=right;

        right.prev=left;

    }

    public int get(int key) {

        if(cache.containsKey(key))

        {

            remove(cache.get(key));

            insert(cache.get(key));

            return cache.get(key).val;

        }

        return -1;

    }

    public void put(int key, int value) {

        if(cache.containsKey(key))

        {

            remove(cache.get(key));

        }

        cache.put(key,new Node(key,value));

        insert(cache.get(key));

        if(cache.size()>capacity)

        {

            Node lru=this.left.next;

            remove(lru);

            cache.remove(lru.key);

        }

    }

    public void insert(Node node)

    {

        Node prev = right.prev;

        Node next = right;

        prev.next=node;

        next.prev=node;

        node.next=next;

        node.prev=prev;

    }

    public void remove(Node node)

    {

        node.next.prev=node.prev;

        node.prev.next=node.next;

    }

}

**14) LFU Cache (Leetcode - 460)**

Design and implement a data structure for a Least Frequently Used (LFU) cache.

Implement the LFUCache class:

LFUCache(int capacity) Initializes the object with the capacity of the data structure.

int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.

void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the least frequently used key before inserting a new item. For this problem, when there is a tie (i.e., two or more keys with the same frequency), the least recently used key would be invalidated.

To determine the least frequently used key, a use counter is maintained for each key in the cache. The key with the smallest use counter is the least frequently used key.

When a key is first inserted into the cache, its use counter is set to 1 (due to the put operation). The use counter for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

Example 1:

Input

["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]

Output

[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation

// cnt(x) = the use counter for key x

// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)

LFUCache lfu = new LFUCache(2);

lfu.put(1, 1); // cache=[1,\_], cnt(1)=1

lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1

lfu.get(1); // return 1

// cache=[1,2], cnt(2)=1, cnt(1)=2

lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.

// cache=[3,1], cnt(3)=1, cnt(1)=2

lfu.get(2); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,1], cnt(3)=2, cnt(1)=2

lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.

// cache=[4,3], cnt(4)=1, cnt(3)=2

lfu.get(1); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,4], cnt(4)=1, cnt(3)=3

lfu.get(4); // return 4

// cache=[4,3], cnt(4)=2, cnt(3)=3

Constraints:

1 <= capacity <= 104

0 <= key <= 105

0 <= value <= 109

At most 2 \* 105 calls will be made to get and put**.**

**Solution :**

class LFUCache {

    HashMap<Integer,DoubleLinkedList> freqMap = new HashMap<>();

    HashMap<Integer,Node> cache = new HashMap<>();

    int capacity=0;

    int minFreq=0;

    int curSize=0;

  public LFUCache(int capacity) {

        this.capacity=capacity;

    }

    public int get(int key) {

        Node cur = cache.get(key);

        if(cur == null)

        {

            return -1;

        }

        update(cur);

        return cur.val;

    }

    public void put(int key, int value) {

         if (capacity == 0) {

            return;

        }

        if(cache.containsKey(key))

        {

            Node cur = cache.get(key);

            cur.val=value;

            update(cur);

        }

        else

        {

            curSize++;

            if(curSize>capacity)

            {

                DoubleLinkedList minFrequency = freqMap.get(minFreq);

                cache.remove(minFrequency.left.next.key);

                minFrequency.delete(minFrequency.left.next);

                curSize--;

            }

            minFreq = 1;

            Node newNode = new Node(key,value);

            DoubleLinkedList curList = freqMap.getOrDefault(1,new DoubleLinkedList());

            curList.insert(newNode);

            freqMap.put(1,curList);

            cache.put(key,newNode);

        }

    }

    public void update(Node cur)

    {

        int curFreq = cur.freq;

        DoubleLinkedList curList = freqMap.get(curFreq);

        curList.delete(cur);

        if(curFreq == minFreq && curList.listSize == 0)

        {

            minFreq++;

        }

        cur.freq++;

        DoubleLinkedList newList = freqMap.getOrDefault(cur.freq,new DoubleLinkedList());

        newList.insert(cur);

        freqMap.put(cur.freq,newList);

    }

    class Node

    {

        int key,val,freq;

        Node next,prev ;

        public Node(int key,int val)

        {

            this.key=key;

            this.val=val;

            this.freq=1;

        }

    }

    class DoubleLinkedList{

        int listSize;

        Node left;

        Node right;

        public DoubleLinkedList()

        {

            this.listSize=0;

            this.left=new Node(0,0);

            this.right=new Node(0,0);

            left.next=right;

            right.prev=left;

        }

        public void insert(Node node)

        {

            Node prev = right.prev;

            prev.next= node;

            right.prev=node;

            node.prev=prev;

            node.next=right;

            listSize++;

        }

        public void delete(Node node)

        {

            node.prev.next= node.next;

            node.next.prev= node.prev;

            listSize--;

        }

    }

}

**15) Swapping Nodes in a Linked List (Leetcode - 1721)**

You are given the head of a linked list, and an integer k.

Return the head of the linked list after swapping the values of the kth node from the beginning and the kth node from the end (the list is 1-indexed).

Example 1:

Input: head = [1,2,3,4,5], k = 2

Output: [1,4,3,2,5]

Example 2:

Input: head = [7,9,6,6,7,8,3,0,9,5], k = 5

Output: [7,9,6,6,8,7,3,0,9,5]

Constraints:

The number of nodes in the list is n.

1 <= k <= n <= 105

0 <= Node.val <= 100

**Solution :**

class Solution {

    public ListNode swapNodes(ListNode head, int k) {

        ListNode left=head;

        for(int i=0;i<k-1;i++)

        {

            left=left.next;

        }

        ListNode cur=left;

        ListNode right=head;

        while(cur.next!=null)

        {

            cur=cur.next;

            right=right.next;

        }

        int temp=left.val;

        left.val=right.val;

        right.val=temp;

        return head;

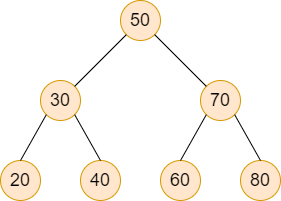
    }

}

**16) BST to sorted DLL (coding Ninja)**

You are provided with a Binary Search Tree (BST), all you have to do is to convert it into the sorted doubly linked list (DLL).

For Example:



Consider the above BST, it will be converted into the below sorted DLL.

Example

Consider the above BST, it will be converted into the below sorted DLL.

Here, 20 is the head node and 80 is the tail node.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

1 <= T <= 10

0 <= N <= 10^4

-10^5 <= DATA <= 10^5

Time Limit: 1sec

**Solution :**

public class Solution

{

    TreeNode<Integer> head,prev;

    public void convert(TreeNode<Integer> root)

    {

        if(root==null) return ;

        convert(root.left);

        if(prev==null)

        {

            head=root;

        }

        else{

            root.left=prev;

            prev.right=root;

        }

        prev=root;

        convert(root.right);

    }

    public static TreeNode<Integer> bstToSortedDLL(TreeNode<Integer> root) {

        // Write your code here.

        Solution s = new Solution();

        s.convert(root);

        return s.head;

    }

}

**Stack**

**1) Remove Outermost Parentheses (Leetcode - 1021) (Easy)**

A valid parentheses string is either empty "", "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation.

For example, "", "()", "(())()", and "(()(()))" are all valid parentheses strings.

A valid parentheses string s is primitive if it is nonempty, and there does not exist a way to split it into s = A + B, with A and B nonempty valid parentheses strings.

Given a valid parentheses string s, consider its primitive decomposition: s = P1 + P2 + ... + Pk, where Pi are primitive valid parentheses strings.

Return s after removing the outermost parentheses of every primitive string in the primitive decomposition of s.

Example 1:

Input: s = "(()())(())"

Output: "()()()"

Explanation:

The input string is "(()())(())", with primitive decomposition "(()())" + "(())".

After removing outer parentheses of each part, this is "()()" + "()" = "()()()".

Example 2:

Input: s = "(()())(())(()(()))"

Output: "()()()()(())"

Explanation:

The input string is "(()())(())(()(()))", with primitive decomposition "(()())" + "(())" + "(()(()))".

After removing outer parentheses of each part, this is "()()" + "()" + "()(())" = "()()()()(())".

Example 3:

Input: s = "()()"

Output: ""

Explanation:

The input string is "()()", with primitive decomposition "()" + "()".

After removing outer parentheses of each part, this is "" + "" = "".

Constraints:

1 <= s.length <= 105

s[i] is either '(' or ')'.

s is a valid parentheses string.

**Solution :**

class Solution {

    public String removeOuterParentheses(String s) {

        Stack<Character> bracket = new Stack<>();

        StringBuilder sb = new StringBuilder("");

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

            if(s.charAt(i)=='('){

                if(bracket.size()>0){

                    sb.append(s.charAt(i));

                }

                bracket.push(s.charAt(i));

            }else{

                bracket.pop();

                if(bracket.size()>0){

                    sb.append(s.charAt(i));

                }

            }

        }

        return sb.toString();

    }

}

**2) Final Prices With a Special Discount in a Shop (Leetcode - 1475)**

You are given an integer array prices where prices[i] is the price of the ith item in a shop.

There is a special discount for items in the shop. If you buy the ith item, then you will receive a discount equivalent to prices[j] where j is the minimum index such that j > i and prices[j] <= prices[i]. Otherwise, you will not receive any discount at all.

Return an integer array answer where answer[i] is the final price you will pay for the ith item of the shop, considering the special discount.

Example 1:

Input: prices = [8,4,6,2,3]

Output: [4,2,4,2,3]

Explanation:

For item 0 with price[0]=8 you will receive a discount equivalent to prices[1]=4, therefore, the final price you will pay is 8 - 4 = 4.

For item 1 with price[1]=4 you will receive a discount equivalent to prices[3]=2, therefore, the final price you will pay is 4 - 2 = 2.

For item 2 with price[2]=6 you will receive a discount equivalent to prices[3]=2, therefore, the final price you will pay is 6 - 2 = 4.

For items 3 and 4 you will not receive any discount at all.

Example 2:

Input: prices = [1,2,3,4,5]

Output: [1,2,3,4,5]

Explanation: In this case, for all items, you will not receive any discount at all.

Example 3:

Input: prices = [10,1,1,6]

Output: [9,0,1,6]

Constraints:

1 <= prices.length <= 500

1 <= prices[i] <= 1000

**Solution :**

class Solution {

    public int[] finalPrices(int[] prices) {

        Stack<Integer> stack = new Stack<>();

        int[] nse =  new int[prices.length];

        for (int i = prices.length-1 ; i >= 0; i--) {

            while (!stack.isEmpty() && stack.peek() > prices[i]) {

                stack.pop();

            }

             if(!stack.isEmpty()) {

                nse[i] = prices[i]-stack.peek();

            }

            else

            {

                nse[i]=prices[i];

            }

            stack.push(prices[i]);

        }

        return nse;

    }

}

**Solution :**

class Solution {

    public int[] finalPrices(int[] prices) {

        Stack<Integer> stack = new Stack<>();

        int[] nse =  new int[prices.length];

        for (int i = prices.length-1 ; i >= 0; i--) {

            while (!stack.isEmpty() && stack.peek() > prices[i]) {

                stack.pop();

            }

             if(!stack.isEmpty()) {

                nse[i] = prices[i]-stack.peek();

            }

            else

            {

                nse[i]=prices[i];

            }

            stack.push(prices[i]);

        }

        return nse;

    }

}

**3) Next Greater Element I (Leetcode - 496)**

The next greater element of some element x in an array is the first greater element that is to the right of x in the same array.

You are given two distinct 0-indexed integer arrays nums1 and nums2, where nums1 is a subset of nums2.

For each 0 <= i < nums1.length, find the index j such that nums1[i] == nums2[j] and determine the next greater element of nums2[j] in nums2. If there is no next greater element, then the answer for this query is -1.

Return an array ans of length nums1.length such that ans[i] is the next greater element as described above.

Example 1:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2]

Output: [-1,3,-1]

Explanation: The next greater element for each value of nums1 is as follows:

- 4 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.

- 1 is underlined in nums2 = [1,3,4,2]. The next greater element is 3.

- 2 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.

Example 2:

Input: nums1 = [2,4], nums2 = [1,2,3,4]

Output: [3,-1]

Explanation: The next greater element for each value of nums1 is as follows:

- 2 is underlined in nums2 = [1,2,3,4]. The next greater element is 3.

- 4 is underlined in nums2 = [1,2,3,4]. There is no next greater element, so the answer is -1.

Constraints:

1 <= nums1.length <= nums2.length <= 1000

0 <= nums1[i], nums2[i] <= 104

All integers in nums1 and nums2 are unique.

All the integers of nums1 also appear in nums2.

Solution :

class Solution {

    public int[] nextGreaterElement(int[] nums1, int[] nums2) {

        Stack<Integer> stack = new Stack<>();

        HashMap<Integer, Integer> map = new HashMap<>();

        for(int i=nums2.length-1;i>=0;i--)

        {

            while(!stack.isEmpty() && nums2[i]>=stack.peek())

            {

                stack.pop();

            }

           if(stack.isEmpty()) map.put(nums2[i],-1);

            else map.put(nums2[i],stack.peek());

            stack.push(nums2[i]);

        }

        System.out.println(map);

        for(int i=0;i<nums1.length;i++)

        {

            nums1[i]=map.getOrDefault(nums1[i],-1);

        }

        return nums1;

    }

}

**4) Implement Stack using Queues (Leetcode - 225)**

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

void push(int x) Pushes element x to the top of the stack.

int pop() Removes the element on the top of the stack and returns it.

int top() Returns the element on the top of the stack.

boolean empty() Returns true if the stack is empty, false otherwise.

Notes:

You must use only standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.

Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

Input

["MyStack", "push", "push", "top", "pop", "empty"]

[[], [1], [2], [], [], []]

Output

[null, null, null, 2, 2, false]

Explanation

MyStack myStack = new MyStack();

myStack.push(1);

myStack.push(2);

myStack.top(); // return 2

myStack.pop(); // return 2

myStack.empty(); // return False

Constraints:

1 <= x <= 9

At most 100 calls will be made to push, pop, top, and empty.

All the calls to pop and top are valid.

**Solution :**

class MyStack {

    private Queue<Integer> q;

    public MyStack() {

        q=new LinkedList<>();

    }

    public void push(int x) {

        q.add(x);

        for(int i=1;i<q.size();i++)

        {

            q.add(q.remove());

        }

    }

    public int pop() {

        return q.remove();

    }

    public int top() {

        return q.peek();

    }

    public boolean empty() {

        return q.isEmpty();

    }

}

**5) Valid Parentheses (Leetcode - 20)**

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

Open brackets must be closed by the same type of brackets.

Open brackets must be closed in the correct order.

Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{}"

Output: true

Example 3:

Input: s = "(]"

Output: false

Constraints:

1 <= s.length <= 104

s consists of parentheses only '()[]{}'.

**Solution :**

class Solution {

    public boolean isValid(String s) {

        Stack<Character> stack = new Stack<Character>(); // create an empty stack

        for (char c : s.toCharArray()) { // loop through each character in the string

            if (c == '(') // if the character is an opening parenthesis

                stack.push(')'); // push the corresponding closing parenthesis onto the stack

            else if (c == '{') // if the character is an opening brace

                stack.push('}'); // push the corresponding closing brace onto the stack

            else if (c == '[') // if the character is an opening bracket

                stack.push(']'); // push the corresponding closing bracket onto the stack

            else if (stack.isEmpty() || stack.pop() != c) // if the character is a closing bracket

                // if the stack is empty (i.e., there is no matching opening bracket) or the top of the stack

                // does not match the closing bracket, the string is not valid, so return false

                return false;

        }

        // if the stack is empty, all opening brackets have been matched with their corresponding closing brackets,

        // so the string is valid, otherwise, there are unmatched opening brackets, so return false

        return stack.isEmpty();

    }

}

**1) Min Stack (Leetcode - 155) (Medium)**

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.

void push(int val) pushes the element val onto the stack.

void pop() removes the element on the top of the stack.

int top() gets the top element of the stack.

int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

Output

[null,null,null,null,-3,null,0,-2]

Explanation

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); // return -3

minStack.pop();

minStack.top(); // return 0

minStack.getMin(); // return -2

Constraints:

-231 <= val <= 231 - 1

Methods pop, top and getMin operations will always be called on non-empty stacks.

At most 3 \* 104 calls will be made to push, pop, top, and getMin.

**Solution :**

class MinStack {

    private static Stack<Integer> stack;

    private static Stack<Integer> min;

    public MinStack()

    {

        min=new Stack<>();

        stack = new Stack<>();

    }

    public void push(int val) {

        if(min.isEmpty() || val<=min.peek())

        {

            min.push(val);

        }

        stack.push(val);

    }

    public void pop() {

        int i=stack.peek();

        int j=min.peek();

        if(i==j)

        {

            min.pop();

        }

        stack.pop();

    }

    public int top() {

        return stack.peek();

    }

    public int getMin() {

        return min.peek();

    }}

**2) Evaluate Reverse Polish Notation (Leetcode - 150)**

You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Note that:

The valid operators are '+', '-', '\*', and '/'.

Each operand may be an integer or another expression.

The division between two integers always truncates toward zero.

There will not be any division by zero.

The input represents a valid arithmetic expression in a reverse polish notation.

The answer and all the intermediate calculations can be represented in a 32-bit integer.

Example 1:

Input: tokens = ["2","1","+","3","\*"]

Output: 9

Explanation: ((2 + 1) \* 3) = 9

Example 2:

Input: tokens = ["4","13","5","/","+"]

Output: 6

Explanation: (4 + (13 / 5)) = 6

Example 3:

Input: tokens = ["10","6","9","3","+","-11","\*","/","\*","17","+","5","+"]

Output: 22

Explanation: ((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5

= ((10 \* (6 / (12 \* -11))) + 17) + 5

= ((10 \* (6 / -132)) + 17) + 5

= ((10 \* 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

Constraints:

1 <= tokens.length <= 104

tokens[i] is either an operator: "+", "-", "\*", or "/", or an integer in the range [-200, 200].

**Solution :**

class Solution {

     boolean isOperator(String op){

        return (op.equals("+") || op.equals("-") || op.equals("\*") || op.equals("/"));

    }

    public int evalRPN(String[] tokens) {

        Stack<Integer> s = new Stack<>();

        int ans=0;

        for(String i:tokens)

        {

            if(isOperator(i))

            {

                int n=s.pop();

                int n1=s.pop();

                if(i.equals("+"))

                {

                    s.push(n1+n);

                }

                else if(i.equals("-"))

                {

                    s.push(n1-n);

                }

                else if(i.equals("/"))

                {

                    s.push(n1/n);

                }

                else if(i.equals("\*"))

                {

                    s.push(n\*n1);

                }

            }

            else

            {

                s.push(Integer.parseInt(i));

            }

        }

        return s.peek();

    }

}

**3) Decode String (Leetcode - 394)**

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].

The test cases are generated so that the length of the output will never exceed 105.

Example 1:

Input: s = "3[a]2[bc]"

Output: "aaabcbc"

Example 2:

Input: s = "3[a2[c]]"

Output: "accaccacc"

Example 3:

Input: s = "2[abc]3[cd]ef"

Output: "abcabccdcdcdef"

Constraints:

1 <= s.length <= 30

s consists of lowercase English letters, digits, and square brackets '[]'.

s is guaranteed to be a valid input.

All the integers in s are in the range [1, 300].

**Solution :**

class Solution {

    public String decodeString(String s) {

        Stack<String> stack = new Stack<>();

        for(char i:s.toCharArray())

        {

            if(i!=']')

            {

                stack.push(i+"");

            }

            else

            {

                // subString upto "["

                String subString="";

                while(!stack.peek().equals("["))

                {

                    subString=stack.pop()+subString;;

                }

                stack.pop();

                //to find number

                String number="";

                while(!stack.isEmpty() && Character.isDigit(stack.peek().charAt(0)))

                {

                    number=stack.pop()+number;

                }

                //number \* subString

                String subResult="";

                for(int j=0;j<Integer.parseInt(number);j++)

                {

                    subResult+=subString;

                }

                stack.push(subResult);

            }

        }

        String res="";

        while(!stack.isEmpty())

        {

            res=stack.pop()+res;

        }

        return res;

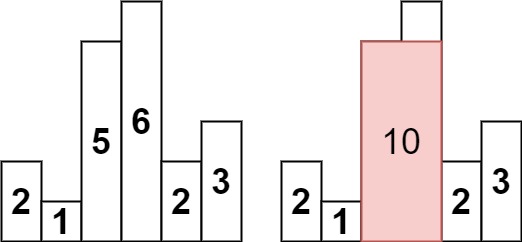
}

}

**1) Largest Rectangle in Histogram (Leetcode - 84) (Hard)**

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1:



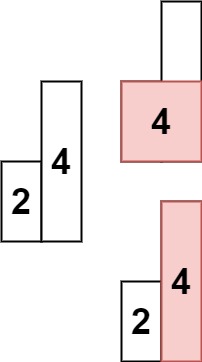
Input: heights = [2,1,5,6,2,3]

Output: 10

Explanation: The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

Example 2:



Input: heights = [2,4]

Output: 4

Constraints:

1 <= heights.length <= 105

0 <= heights[i] <= 104

**Solution : (NSE,GSE,PSE,PGE)**

class Solution {

    Stack<Integer> stack;

    public int[] nextSmallerElement(int[] heights)

    {

        int[] nse=new int[heights.length];

        stack=new Stack<>();

        for (int i = heights.length-1 ; i >= 0; i--) {

            while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]) {

                stack.pop();

            }

            if (stack.isEmpty()) {

                nse[i] = heights.length;

            } else {

                nse[i] = stack.peek();

            }

            stack.push(i);

        }

        return nse;

    }

    public int[] nextGreaterElement(int[] heights)

    {

        int[] gse = new int[heights.length];

        stack = new Stack<Integer>();

        for(int i=heights.length-1;i>=0;i--)

        {

            while(!stack.isEmpty() && heights[i]>=heights[stack.peek()])

            {

                stack.pop();

            }

            gse[i]=(!stack.isEmpty())?stack.peek():-1;

            stack.push(i);

            System.out.println(stack);

        }

        return gse;

    }

    public int[] previousGreaterElement(int[] heights)

    {

        int pge[] = new int[heights.length];

        stack = new Stack<Integer>();

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

        {

            while(!stack.isEmpty() && heights[i]>=heights[stack.peek()])

            {

                stack.pop();

            }

            pge[i]=(!stack.isEmpty())?stack.peek():-1;

            stack.push(i);

        }

        return pge;

    }

     public int[] previousSmallerElement(int[] heights)

    {

        int pse[]= new int[heights.length];

        stack = new Stack<Integer>();

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

            while (!stack.isEmpty() && heights[stack.peek()] >= heights[i]) {

                stack.pop();

            }

            if (stack.isEmpty()) {

                pse[i] = -1;

            } else {

                pse[i] = stack.peek();

            }

            stack.push(i);

        }

        return pse;

    }

    public int largestRectangleArea(int[] heights) {

        int[] nse=nextSmallerElement(heights);

        int[] pse=previousSmallerElement(heights);

        int max=Integer.MIN\_VALUE;

        System.out.println(Arrays.toString(nse)+" "+Arrays.toString(pse));

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

        {

            int area=(nse[i]-pse[i]-1)\*heights[i];

            max=Math.max(max,area);

        }

        return max;

    }

}

**2) Longest Valid Parentheses (Leetcode – 32)**

Given a string containing just the characters '(' and ')', return the length of the longest valid (well-formed) parentheses

substring.

Example 1:

Input: s = "(()"

Output: 2

Explanation: The longest valid parentheses substring is "()".

Example 2:

Input: s = ")()())"

Output: 4

Explanation: The longest valid parentheses substring is "()()".

Example 3:

Input: s = ""

Output: 0

Constraints:

0 <= s.length <= 3 \* 104

s[i] is '(', or ')'.

**Solution :**

class Solution {

    public int longestValidParentheses(String s) {

        int max=0;

        Stack<Integer> stack = new Stack<Integer>();

        stack.push(-1);

        for(int i=0;i<s.length();i++)

        {

            if(s.charAt(i)=='(')

            {

                stack.push(i);

            }

            else

            {

                stack.pop();

                if(stack.isEmpty())

                {

                    stack.push(i);

                }

                max=Math.max(max,i-stack.peek());

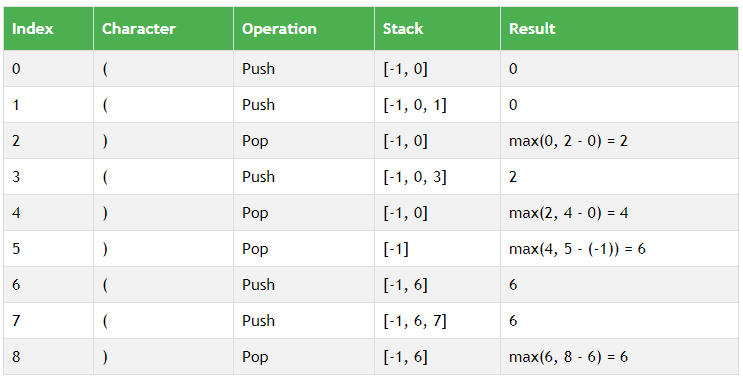
            }

        }

     return max;

    }

}



**Queue**

**1) Find the Winner of the Circular Game (Leetcode - 1823) (Medium)**

There are n friends that are playing a game. The friends are sitting in a circle and are numbered from 1 to n in clockwise order. More formally, moving clockwise from the ith friend brings you to the (i+1)th friend for 1 <= i < n, and moving clockwise from the nth friend brings you to the 1st friend.

The rules of the game are as follows:

Start at the 1st friend.

Count the next k friends in the clockwise direction including the friend you started at. The counting wraps around the circle and may count some friends more than once.

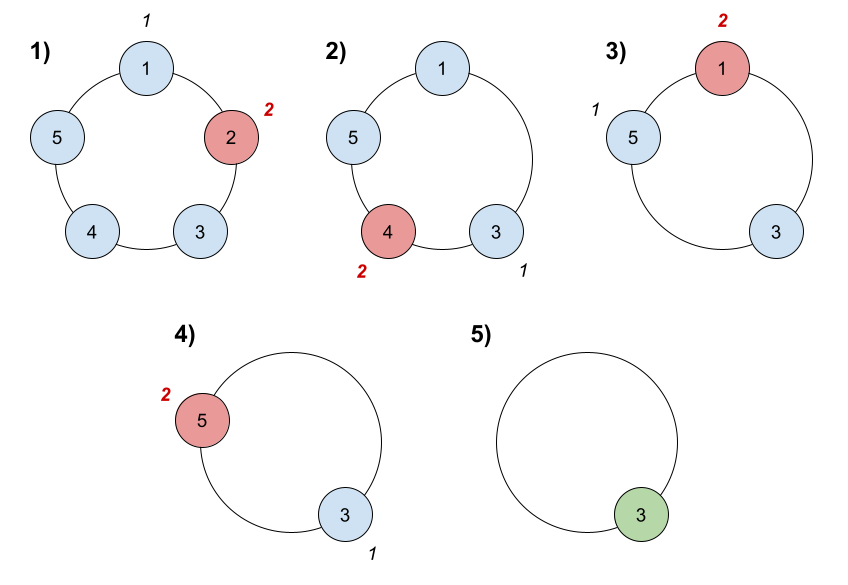
The last friend you counted leaves the circle and loses the game.

If there is still more than one friend in the circle, go back to step 2 starting from the friend immediately clockwise of the friend who just lost and repeat.

Else, the last friend in the circle wins the game.

Given the number of friends, n, and an integer k, return the winner of the game.

Example 1:



Input: n = 5, k = 2

Output: 3

Explanation: Here are the steps of the game:

1) Start at friend 1.

2) Count 2 friends clockwise, which are friends 1 and 2.

3) Friend 2 leaves the circle. Next start is friend 3.

4) Count 2 friends clockwise, which are friends 3 and 4.

5) Friend 4 leaves the circle. Next start is friend 5.

6) Count 2 friends clockwise, which are friends 5 and 1.

7) Friend 1 leaves the circle. Next start is friend 3.

8) Count 2 friends clockwise, which are friends 3 and 5.

9) Friend 5 leaves the circle. Only friend 3 is left, so they are the winner.

Example 2:

Input: n = 6, k = 5

Output: 1

Explanation: The friends leave in this order: 5, 4, 6, 2, 3. The winner is friend 1.

Constraints:

1 <= k <= n <= 500

**Solution :**

class Solution {

    public int findTheWinner(int n, int k) {

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

        for(int i=1;i<=n;i++)

        {

            list.add(i);

        }

        int remove=0;

        while(list.size()>1)

        {

            remove=(remove-1+k)%list.size();

            list.remove(remove);

        }

        return list.get(0);

    }

}

**1) Sliding Window Maximum (Leetcode - 239) (Hard)**

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [3,3,5,5,6,7]

Explanation:

Window position Max

--------------- -----

[1 3 -1] -3 5 3 6 7 3

1 [3 -1 -3] 5 3 6 7 3

1 3 [-1 -3 5] 3 6 7 5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6] 7 6

1 3 -1 -3 5 [3 6 7] 7

Example 2:

Input: nums = [1], k = 1

Output: [1]

Constraints:

1 <= nums.length <= 105

-104 <= nums[i] <= 104

1 <= k <= nums.length

**Solution :**

class Solution {

    public int[] maxSlidingWindow(int[] nums, int k) {

        int [] max = new int[nums.length-k+1];

        int j=0;

        Deque<Integer> deq = new ArrayDeque<>();

        for(int i=0;i<k;i++){

            while(!deq.isEmpty() && nums[i]>deq.peekLast())

            {

                deq.pollLast();

            }

            deq.addLast(nums[i]);

        }

        max[j]=deq.peekFirst();

        j++;

        for(int i=k;i<nums.length;i++)

        {

            if(nums[i-k]==deq.peekFirst())

            {

                deq.pollFirst();

            }

            while(!deq.isEmpty() && nums[i]>deq.peekLast())

            {

                deq.pollLast();

            }

            deq.addLast(nums[i]);

            max[j]=deq.peekFirst();

            j++;

        }

        return max;

    }

}

**PRIORITY QUEUE**

**1) Maximum Product of Two Elements in an Array (Leetcode 1464) (Easy)**

Given the array of integers nums, you will choose two different indices i and j of that array. Return the maximum value of (nums[i]-1)\*(nums[j]-1).

Example 1:

Input: nums = [3,4,5,2]

Output: 12

Explanation: If you choose the indices i=1 and j=2 (indexed from 0), you will get the maximum value, that is, (nums[1]-1)\*(nums[2]-1) = (4-1)\*(5-1) = 3\*4 = 12.

Example 2:

Input: nums = [1,5,4,5]

Output: 16

Explanation: Choosing the indices i=1 and j=3 (indexed from 0), you will get the maximum value of (5-1)\*(5-1) = 16.

Example 3:

Input: nums = [3,7]

Output: 12

Constraints:

2 <= nums.length <= 500

1 <= nums[i] <= 10^3

**Solution :**

class Solution {

    public int maxProduct(int[] nums) {

        PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());

        for(int i:nums)

        {

            pq.offer(i);

        }

        return (pq.poll()-1)\*(pq.poll()-1);

    }

}

**2) Minimum Number Game(Leetcode -2974)**

You are given a 0-indexed integer array nums of even length and there is also an empty array arr. Alice and Bob decided to play a game where in every round Alice and Bob will do one move. The rules of the game are as follows:

Every round, first Alice will remove the minimum element from nums, and then Bob does the same.

Now, first Bob will append the removed element in the array arr, and then Alice does the same.

The game continues until nums becomes empty.

Return the resulting array arr.

Example 1:

Input: nums = [5,4,2,3]

Output: [3,2,5,4]

Explanation: In round one, first Alice removes 2 and then Bob removes 3. Then in arr firstly Bob appends 3 and then Alice appends 2. So arr = [3,2].

At the begining of round two, nums = [5,4]. Now, first Alice removes 4 and then Bob removes 5. Then both append in arr which becomes [3,2,5,4].

Example 2:

Input: nums = [2,5]

Output: [5,2]

Explanation: In round one, first Alice removes 2 and then Bob removes 5. Then in arr firstly Bob appends and then Alice appends. So arr = [5,2].

Constraints:

2 <= nums.length <= 100

1 <= nums[i] <= 100

nums.length % 2 == 0

**Solution :**

class Solution {

    public int[] numberGame(int[] nums) {

        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        int[] res = new int[nums.length];

        int i = 0;

        for(int num: nums) minHeap.add(num);

        while(minHeap.size() >= 2) {

            int alice = minHeap.poll();

            int bob = minHeap.poll();

            res[i++] = bob;

            res[i++] = alice;

        }

        if(!minHeap.isEmpty()) {

            res[i++] = minHeap.poll();

        }

        return res;

    }

}

**1) Kth Largest Element in an Array (Leetcode - 215) (Medium)**

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2

Output: 5

Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4

Output: 4

Constraints:

1 <= k <= nums.length <= 105

-104 <= nums[i] <= 104

**Solution :**

class Solution {

    public int findKthLargest(int[] nums, int k) {

        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for(int i:nums)

        {

            if(pq.size()<k)

            {

                pq.offer(i);

            }

            else

            {

                pq.offer(i);

                pq.poll();

            }

        }

        return pq.poll();

    }

}

**2) Top K Frequent Elements(Leetcode - 347)**

Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Example 1:

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

Example 2:

Input: nums = [1], k = 1

Output: [1]

Constraints:

1 <= nums.length <= 105

-104 <= nums[i] <= 104

k is in the range [1, the number of unique elements in the array].

It is guaranteed that the answer is unique.

Follow up: Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

**Solution :**

class Solution {

    public int[] topKFrequent(int[] nums, int k) {

        Map<Integer,Integer> map = new HashMap<>();

        for(int i:nums)

        {

            map.put(i,map.getOrDefault(i,0)+1);

        }

        PriorityQueue<Map.Entry<Integer,Integer>> pq = new PriorityQueue<>((a,b)->a.getValue()-b.getValue());

        for(Map.Entry<Integer,Integer> it:map.entrySet())

        {

            pq.add(it);

            if(pq.size()>k)

            {

                pq.poll();

            }

        }

        int[] arr = new int[k];

        int i=0;

        while(!pq.isEmpty())

        {

            arr[i]=pq.poll().getKey();

            i++;

        }

        return arr;

    }

}

**1) Merge k Sorted Lists (Leetcode - 23) (Hard)**

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

Explanation: The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

Example 2:

Input: lists = []

Output: []

Example 3:

Input: lists = [[]]

Output: []

Constraints:

k == lists.length

0 <= k <= 104

0 <= lists[i].length <= 500

-104 <= lists[i][j] <= 104

lists[i] is sorted in ascending order.

The sum of lists[i].length will not exceed 104.

**Solution :**

class Solution {

    public ListNode mergeKLists(ListNode[] lists) {

        PriorityQueue<ListNode> pq = new PriorityQueue<>((a,b)->  a.val-b.val);

        for(ListNode l : lists)

        {

            if(l!=null) pq.offer(l);

        }

        ListNode dummy = new ListNode(0);

        ListNode temp = dummy;

        while(!pq.isEmpty())

        {

            ListNode node = pq.poll();

            temp.next=node;

            temp=temp.next;

            if(temp.next!=null)//node.next

            {

                pq.offer(temp.next);

            }

        }

        return dummy.next;

    }

}

**Trie**

**1) Implement Trie (Prefix Tree) (Leetcode - 208)**

A trie (pronounced as "try") or prefix tree is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

Trie() Initializes the trie object.

void insert(String word) Inserts the string word into the trie.

boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.

boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

Example 1:

Input

["Trie", "insert", "search", "search", "startsWith", "insert", "search"]

[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]

Output

[null, null, true, false, true, null, true]

Explanation

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // return True

trie.search("app"); // return False

trie.startsWith("app"); // return True

trie.insert("app");

trie.search("app"); // return True

Constraints:

1 <= word.length, prefix.length <= 2000

word and prefix consist only of lowercase English letters.

At most 3 \* 104 calls in total will be made to insert, search, and startsWith.

**Solution :**

class Trie {

    class TrieNode

    {

        TrieNode[] child;

        boolean isEnd;

        TrieNode()

        {

            child = new TrieNode[26];

        }

    }

    TrieNode root;

    public Trie() {

        root=new TrieNode();

    }

    public void insert(String word) {

        TrieNode temp = root;

        for(char c:word.toCharArray())

        {

            int idx=c-'a';

            if(temp.child[idx]==null)

            {

                temp.child[idx]=new TrieNode();

            }

            temp=temp.child[idx];

        }

        temp.isEnd=true;

    }

    public boolean search(String word) {

        TrieNode temp = root;

        for(char c:word.toCharArray())

        {

            int idx=c-'a';

            if(temp.child[idx]==null)

            {

                return false;

            }

            temp=temp.child[idx];

        }

        if(temp.isEnd) return true;

        return false;

    }

    public boolean startsWith(String prefix) {

        TrieNode temp = root;

        for(char c:prefix.toCharArray())

        {

            int idx=c-'a';

            if(temp.child[idx]==null)

            {

                return false;

            }

            temp=temp.child[idx];

        }

        return true;

    }

}

**1) Longest Common Prefix(Leetcode - 14) (Easy)**

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]

Output: "fl"

Example 2:

Input: strs = ["dog","racecar","car"]

Output: ""

Explanation: There is no common prefix among the input strings.

Constraints:

1 <= strs.length <= 200

0 <= strs[i].length <= 200

strs[i] consists of only lowercase English letters.

**Solution :**

class Solution {

    public String longestCommonPrefix(String[] strs) {

        Trie trie = new Trie();

        for(String i:strs)

        {

            trie.insert(i);

        }

        return trie.longestCommonPrefix();

    }

    class TrieNode

    {

        TrieNode[] child;

        boolean isEnd;

        TrieNode()

        {

            child = new TrieNode[26];

            isEnd=false;

        }

    }

    class Trie

    {

        TrieNode root;

        Trie()

        {

            root=new TrieNode();

        }

        public void insert(String s)

        {

            TrieNode temp=root;

            for(char c:s.toCharArray())

            {

                int idx=c-'a';

                if(temp.child[idx]==null)

                {

                    temp.child[idx]=new TrieNode();

                }

                temp=temp.child[idx];

            }

            temp.isEnd=true;

        }

        public String longestCommonPrefix()

        {

            StringBuilder s = new StringBuilder();

            TrieNode temp=root;

            while(true)

            {

                int count = 0;

            int idx=-1;

                for(int i=0;i<26;i++)

                {

                    if(temp.child[i]!=null)

                    {

                        count++;

                        idx=i;

                    }

                }

                if(count==1 && !temp.isEnd)

                {

                    s.append((char)(idx+'a'));

                    temp=temp.child[idx];

                }

                else

                {

                    break;

                }

            }

            return s.toString();

        }

    }

}

**2) Maximum XOR (Leetcode -2932 )**

Given an array of positive integers, find 2 elements such that their xor: a ^ b is maximum.

Input Format

The first line of input contains T - number of test cases. It's followed by 2T lines - the first line contains N: the size of the array. The second line contains N integers - the elements of the array.

Output Format

For each test case, print the value of the maximum xor, separated by new line.

Constraints

30 points

2 <= N <= 103

120 points

2 <= N <= 105

General Constraints

1 <= T <= 100

0 <= A[i] <= 106

Example

Input

2

3

12 15 9

5

13 11 35 3 32

Output

6

46

Explanation

Example 1:

You can form the following xor pairs:

12^15 = 3, 12^9 = 5, 15^9 = 6 : max = 6

**Solution :**

import java.io.\*;

import java.util.\*;

class TrieNode

    {

        TrieNode child[];

        TrieNode()

        {

            child=new TrieNode[2];

        }

    }

     class Trie

    {

        TrieNode root;

        Trie()

        {

            root = new TrieNode();

        }

        public void insert(int ele)

        {

            TrieNode temp=root;

            for(int i=31;i>=0;i--)

            {

                int idx = (ele>>i)&1;

                if(temp.child[idx]==null)

                {

                    temp.child[idx]=new TrieNode();

                }

                temp=temp.child[idx];

            }

        }

        public int maxXor(int ele)

        {

            TrieNode temp = root;

            int x=0;

            for(int i=31;i>=0;i--)

            {

                int idx=(ele>>i)&1;

                if(temp.child[1-idx]!=null)

                {

                    temp=temp.child[1-idx];

                    x=x+(1<<i);

                }

                else

                {

                    temp=temp.child[idx];

                }

            }

            return x;

        }

    }

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int t = sc.nextInt();

        for(int i=0;i<t;i++)

        {

            Trie trie = new Trie();

            int n = sc.nextInt();

            int a[] = new int[n];

            for(int j=0;j<n;j++)

            {

                a[j]=sc.nextInt();

                trie.insert(a[j]);

            }

            int max=0;

            for(int k:a)

            {

                max=Math.max(max,trie.maxXor(k));

            }

            System.out.println(max);

        }

    }

}

**3) Max Rhyming Word Length**

Given is a wordlist L, and a word W. Your task is to find the length of the longest word in L having the longest common suffix with W.

Input Format

The first line of input contains N - the length of the list of words. The next N lines contain a single word having lowercase English alphabets. The next line contains Q - number of queries. Each of the next Q lines contains a single word W having lowercase English alphabets.

Output Format

For each query W, print the length of the longest word having the longest common suffix with W, separated by a newline.

Constraints

50 points

1 <= N, Q <= 1000

1 <= len(word) <= 100

100 points

1 <= N, Q <= 10000

1 <= len(word) <= 500

Example

Input

4

crime

fast

time

cast

3

dime

gist

algorithm

Output

5

4

0

Explanation

Example 1:

Both the words crime and time have the longest common suffix with dime. Hence, the length of the longest word is 5.

Example 2:

Both the words fast and cast have the longest common suffix with gist. Hence, the length of the longest word is 4.

Example 3:

There is no word in L having a common suffix with algorithm. Hence, the length of the longest word is 0.

**Solution :**

import java.io.\*;

import java.util.\*;

class TrieNode

{

    TrieNode child[];

    int lcs;

    TrieNode()

    {

            child=new TrieNode[26];

            lcs=0;

    }

}

class Trie

{

        TrieNode root;

        Trie()

        {

            root = new TrieNode();

        }

        public void insert(String ele)

        {

            TrieNode temp=root;

            for(int i=ele.length()-1;i>=0;i--)

            {

                int idx=ele.charAt(i)-'a';

                if(temp.child[idx]==null)

                {

                    temp.child[idx]=new TrieNode();

                }

                temp=temp.child[idx];

                temp.lcs=Math.max(temp.lcs,ele.length());

            }

        }

        public int lcs(String ele)

        {

            TrieNode temp = root;

            for(int i=ele.length()-1;i>=0;i--)

            {

                int idx=ele.charAt(i)-'a';

                if(temp.child[idx]==null)

                {

                    break;

                }

                else

                {

                    temp=temp.child[idx];

                }

            }

            return temp.lcs;

        }

    }

public class Main {

    public static void main(String[] args) {

        /\* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Main. \*/

        Scanner sc = new Scanner(System.in);

        int t = sc.nextInt();

        Trie  trie = new Trie();

        for(int i=0;i<t;i++)

        {

            trie.insert(sc.next());

        }

        int n=sc.nextInt();

        for(int j=0;j<n;j++)

        {

            System.out.println(trie.lcs(sc.next()));

        }

    }

}