142. Linked List Cycle II

146. LRU Cache

148. Sort List

152. Maximum Product Subarray

155. Min Stack

160. Intersection of Two Linked Lists

169. Majority Element

198. House Robber

200. Number of Islands

206. Reverse Linked List

208. Implement Trie (Prefix Tree)

215. Kth Largest Element in an Array

221. Maximal Square

226. Invert Binary Tree

234. Palindrome Linked List

236. Lowest Common Ancestor of a Binary Tree

238. Product of Array Except Self

239. Sliding Window Maximum

240. Search a 2D Matrix II

279. Perfect Squares

283. Move Zeroes

287. Find the Duplicate Number

297. Serialize and Deserialize Binary Tree

300. Longest Increasing Subsequence

301. Remove Invalid Parentheses

309. Best Time to Buy and Sell Stock with Cooldown

312. Burst Balloons

322. Coin Change

337. House Robber III

338. Counting Bits

347. Top K Frequent Elements

394. Decode String

406. Queue Reconstruction by Height

416. Partition Equal Subset Sum

437. Path Sum III

438. Find All Anagrams in a String

448. Find All Numbers Disappeared in an Array

494. Target Sum

538. Convert BST to Greater Tree

543. Diameter of Binary Tree

560. Subarray Sum Equals K

572. Subtree of Another Tree

581. Shortest Unsorted Continuous Subarray

617. Merge Two Binary Trees

621. Task Scheduler

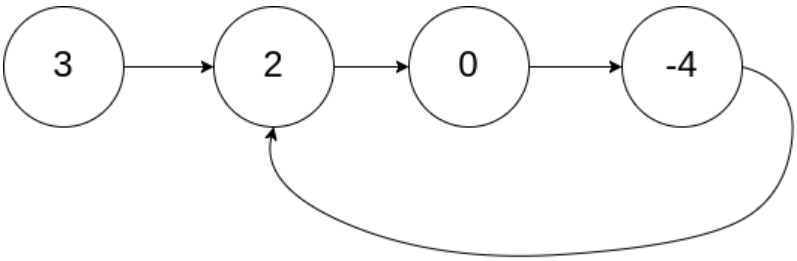
647. Palindromic Substrings

771. Jewels and Stones

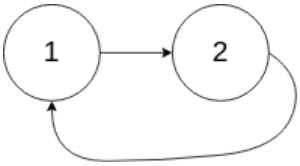
142. Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.
To represent a cycle in the given linked list, we use an integer `pos` which represents the position (0-indexed) in the linked list where tail connects to. If `pos` is `-1`, then there is no cycle in the linked list.
Note: 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.

```java
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head == null || head.next == null){
            return null;
        }
        Set<ListNode> set = new HashSet<>();
        ListNode p = head;
        while(p != null){
            if(!set.add(p)){
                return p;
             }
             p = p.next;
         }
         return null;
     }
}
```

146. LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `put`.
`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`put(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.
Follow up:
Could you do both operations in **O(1)** time complexity?
Example:

LRUCache cache = new LRUCache(2 /* capacity */);

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);   // returns 1
cache.put(3, 3);   // evicts key 2
cache.get(2);   // returns -1 (not found)
cache.put(4, 4);   // evicts key 1
cache.get(1);   // returns -1 (not found)
cache.get(3);   // returns 3
cache.get(4);   // returns 4

```java
class LRUCache extends LinkedHashMap<Integer, Integer>{
    private int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75F, true);
        this.capacity = capacity;
    }

    public int get(int key) {
         return super.getOrDefault(key, -1);
     }

     public void put(int key, int value) {
         super.put(key, value);
     }

     @Override
     protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {
         return size() > capacity;
     }
}
```

```java
import java.util.Hashtable;
public class LRUCache {

  class DLinkedNode {
    int key;
    int value;
    DLinkedNode prev;
    DLinkedNode next;
  }

   private void addNode(DLinkedNode node) {
     /**
      * Always add the new node right after head.
      */
```

```
    node.prev = head;
    node.next = head.next;

    head.next.prev = node;
    head.next = node;
  }

  private void removeNode(DLinkedNode node){
    /**
     * Remove an existing node from the linked list.
     */
    DLinkedNode prev = node.prev;
    DLinkedNode next = node.next;

    prev.next = next;
    next.prev = prev;
  }

  private void moveToHead(DLinkedNode node){
    /**
     * Move certain node in between to the head.
     */
    removeNode(node);
    addNode(node);
  }

  private DLinkedNode popTail() {
    /**
     * Pop the current tail.
     */
    DLinkedNode res = tail.prev;
    removeNode(res);
    return res;
  }

  private Hashtable<Integer, DLinkedNode> cache =
          new Hashtable<Integer, DLinkedNode>();
  private int size;
  private int capacity;
  private DLinkedNode head, tail;

  public LRUCache(int capacity) {
    this.size = 0;
    this.capacity = capacity;

    head = new DLinkedNode();
    // head.prev = null;

```

```
    tail = new DLinkedNode();
    // tail.next = null;

    head.next = tail;
    tail.prev = head;
  }

  public int get(int key) {
    DLinkedNode node = cache.get(key);
    if (node == null) return -1;

    // move the accessed node to the head;
    moveToHead(node);

    return node.value;
  }

  public void put(int key, int value) {
    DLinkedNode node = cache.get(key);

    if(node == null) {
      DLinkedNode newNode = new DLinkedNode();
      newNode.key = key;
      newNode.value = value;

      cache.put(key, newNode);
      addNode(newNode);

      ++size;

      if(size > capacity) {
        // pop the tail
        DLinkedNode tail = popTail();
        cache.remove(tail.key);
        --size;
      }
    } else {
       // update the value.
       node.value = value;
       moveToHead(node);
     }
   }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
```

```
*/
```

148. Sort List

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

Example 1:

Input: 4->2->1->3

Output: 1->2->3->4

Example 2:

Input: -1->5->3->4->0

Output: -1->0->3->4->5

```java
class Solution {
    public ListNode sortList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode slow = head;
        ListNode fast = head.next.next;
        while(slow != null && fast != null && fast.next != null){
            slow = slow.next;
             fast = fast.next.next;
         }
         ListNode node = slow.next;
         slow.next = null;
         ListNode p = sortList(head);
         ListNode q = sortList(node);
         return merge(p, q);
     }

     private ListNode merge(ListNode p, ListNode q){
         if(p == null){
             return q;
         }
         if(q == null){
             return p;
         }
         ListNode dummy = new ListNode(-1);
         ListNode pn = p;
         ListNode qn = q;
         ListNode cur = dummy;
         while(pn != null || qn != null){
             if(pn == null){
                 cur.next = qn;
                 qn = qn.next;
             }else if(qn == null){
                 cur.next = pn;
```

```
                pn = pn.next;
            }else{
                if(pn.val < qn.val){
                    cur.next = pn;
                    pn = pn.next;
                }else{
                    cur.next = qn;
                    qn = qn.next;
                }
            }
            cur = cur.next;
        }
        return dummy.next;
    }
}
```

152. Maximum Product Subarray

Given an integer array `nums`, find the contiguous subarray within an array (containing at least one number) which has the largest product.

Example 1:
Input: [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
Example 2:
Input: [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

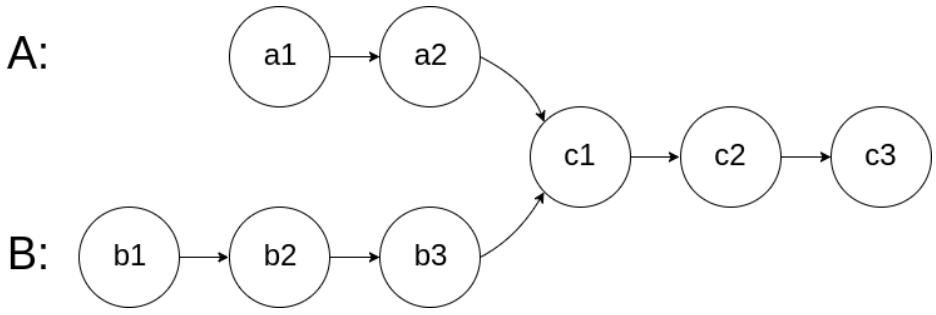
```
public class Solution {
    public int maxProduct(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }
        int max = A[0], min = A[0], result = A[0];
        for (int i = 1; i < A.length; i++) {
            int temp = max;
            max = Math.max(Math.max(max * A[i], min * A[i]), A[i]);
            min = Math.min(Math.min(temp * A[i], min * A[i]), A[i]);
            if (max > result) {
                result = max;
            }
        }
        return result;
    }
}
```

155. Min Stack

```java
class MinStack {
    int min = Integer.MAX_VALUE;
    Stack<Integer> stack = new Stack<Integer>();
    public void push(int x) {
        // only push the old minimum value when the current
        // minimum value changes after pushing the new value x
        if(x <= min){
            stack.push(min);
            min=x;
         }
         stack.push(x);
     }

     public void pop() {
         // if pop operation could result in the changing of the current minimum value,
         // pop twice and change the current minimum value to the last minimum value.
         if(stack.pop() == min) min=stack.pop();
     }

     public int top() {
         return stack.peek();
     }

     public int getMin() {
         return min;
     }
}
```

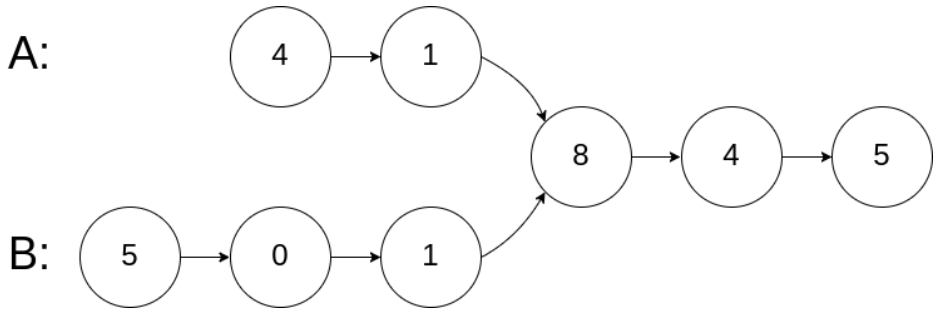
160. Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.
For example, the following two linked lists:



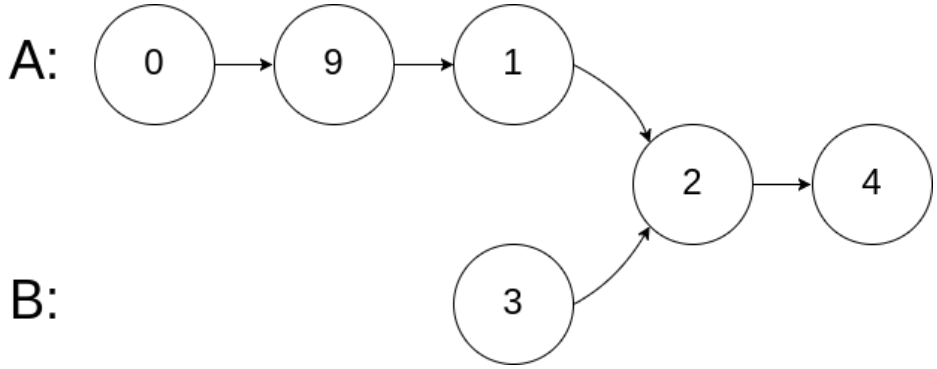
begin to intersect at node c1.

Example 1:



Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
Output: Reference of the node with value = 8
Input 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 [5,0,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

Example 2:



Input: intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
Output: Reference of the node with value = 2
Input 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 [0,9,1,2,4]. 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: null
Input 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.

Notes:

- If the two linked lists have no intersection at all, return `null`.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

```
1 public class Solution {
2     public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
3         if(headA == null || headB == null){
4             return null;
```

```
        }
        Set<ListNode> set = new HashSet<>();
        ListNode p = headA;
        ListNode q = headB;
        while(p != null){
            set.add(p);
            p = p.next;
         }
         while(q != null){
            if(!set.add(q)){
                return q;
            }
            q = q.next;
         }
         return null;
    }
}
```

169. Majority Element

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** `⌊ n/2 ⌋` times.

You may assume that the array is non-empty and the majority element always exist in the array.

Example 1:

Input: [3,2,3]

Output: 3

Example 2:

Input: [2,2,1,1,1,2,2]

Output: 2

```
class Solution {
    public int majorityElement(int[] nums) {
        if(nums == null || nums.length == 0){
            return -1;
        }
        int res = nums[0];
        int count = 1;
        for(int i = 1; i < nums.length;i ++){
            if(count == 0){
                res = nums[i];
                count = 1;
                continue;
            }
            if(res == nums[i]){
                count ++;
            }else{
                count --;

            }
```

```
        }
        return res;
    }
}
```

198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.
Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.
Example 1:
Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
Example 2:
Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.

```
public int rob(int[] num) {
    int[][] dp = new int[num.length + 1][2];
    for (int i = 1; i <= num.length; i++) {
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);
        dp[i][1] = num[i - 1] + dp[i - 1][0];
    }
    return Math.max(dp[num.length][0], dp[num.length][1]);
}
```

dp[i][1] means we rob the current house and dp[i][0] means we don't,
so it is easy to convert this to O(1) space

```
public int rob(int[] num) {
    int prevNo = 0;
    int prevYes = 0;
    for (int n : num) {
        int temp = prevNo;
        prevNo = Math.max(prevNo, prevYes);
        prevYes = n + temp;
    }
    return Math.max(prevNo, prevYes);
}
```

200. Number of Islands

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.
Example 1:
Input:
11110
11010
11000
00000

Output: 1
Example 2:
Input:
11000
11000
00100
00011

Output: 3

```
public class Solution {

private int n;
private int m;

public int numIslands(char[][] grid) {
    int count = 0;
    n = grid.length;
    if (n == 0) return 0;
     m = grid[0].length;
     for (int i = 0; i < n; i++){
         for (int j = 0; j < m; j++)
             if (grid[i][j] == '1') {
                 DFSMarking(grid, i, j);
                 ++count;
             }
     }
     return count;
}

private void DFSMarking(char[][] grid, int i, int j) {
    if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1') return;
    grid[i][j] = '0';
    DFSMarking(grid, i + 1, j);
    DFSMarking(grid, i - 1, j);
    DFSMarking(grid, i, j + 1);
    DFSMarking(grid, i, j - 1);
}

}

```

206. Reverse Linked List

Reverse a singly linked list.
Example:
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL

Follow up:

A linked list can be reversed either iteratively or recursively. Could you implement both?

```java
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        ListNode dummy = new ListNode(-1);
        ListNode p = head;
        while(p != null){
            ListNode next = p.next;
             p.next = dummy.next;
             dummy.next = p;
             p = next;
         }
         return dummy.next;
     }
}
```

208. Implement Trie (Prefix Tree)

Implement a trie with `insert`, `search`, and `startsWith` methods.

Example:

Trie trie = new Trie();

trie.insert("apple");
trie.search("apple"); // returns true
trie.search("app"); // returns false
trie.startsWith("app"); // returns true
trie.insert("app");
trie.search("app"); // returns true

Note:

- You may assume that all inputs are consist of lowercase letters `a-z`.
- All inputs are guaranteed to be non-empty strings.

```java
class Trie {
    private TreeNode root;
    private class TreeNode{
        private TreeNode[] nodes = new TreeNode[26];
        private boolean isEnd;
        private char val;

        public TreeNode(){

         }
         public TreeNode(char c){
            TreeNode treeNode = new TreeNode();
             treeNode.val = c;
         }
     }
```

```
16      /** Initialize your data structure here. */
17      public Trie() {
18          root = new TreeNode();
19      }
20
21      /** Inserts a word into the trie. */
22      public void insert(String word) {
23          if(word == null || word.length() == 0){
24              return;
25          }
26          TreeNode treeNode = root;
27          char[] arr = word.toCharArray();
28          for(int i = 0; i < arr.length; i++){
29              int idx = arr[i] - 'a';
30              if(treeNode.nodes[idx] == null){
31                  treeNode.nodes[idx] = new TreeNode(arr[i]);
32
33              }
34              treeNode = treeNode.nodes[idx];
35          }
36          treeNode.isEnd = true;
37      }
38
39      /** Returns if the word is in the trie. */
40      public boolean search(String word) {
41          if(word == null || word.length() == 0){
42              return false;
43          }
44          TreeNode treeNode = root;
45          char[] arr = word.toCharArray();
46          for(int i = 0; i < arr.length; i++){
47              int idx = arr[i] - 'a';
48              if(treeNode.nodes[idx] == null){
49                  return false;
50              }
51
52              treeNode = treeNode.nodes[idx];
53          }
54          return treeNode.isEnd;
55      }
56
57      /** Returns if there is any word in the trie that starts with the given prefix. */
58      public boolean startsWith(String word) {
59          if(word == null || word.length() == 0){
60              return false;
61          }
62          TreeNode treeNode = root;
63          char[] arr = word.toCharArray();
```

```
        for(int i = 0; i < arr.length; i++){
            int idx = arr[i] - 'a';
            if(treeNode.nodes[idx] == null){
                return false;
            }
            treeNode = treeNode.nodes[idx];
        }
        return true;
    }


}

/**
 * Your Trie object will be instantiated and called as such:
 * Trie obj = new Trie();
 * obj.insert(word);
 * boolean param_2 = obj.search(word);
 * boolean param_3 = obj.startsWith(prefix);
 */
```

215. Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

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

Output: 5

Example 2:

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

Output: 4

Note:

You may assume k is always valid, 1 ≤ k ≤ array's length.

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        return find(nums, 0, nums.length-1,nums.length - k);
    }

    private int find(int[] nums, int start, int end, int k){
        if(start > end || start < 0 || end > nums.length - 1){
            return Integer.MAX_VALUE;
        }
         int pivot = nums[end];
         int left = start;
         for(int i = start; i < end; i++){
            if(nums[i] <= pivot){
               swap(nums, left ++, i);
            }
```

```
        }
        swap(nums, left, end);
        if(left == k){
            return nums[left];
        }else if(left > k){
            return find(nums, start, left - 1, k);
        }else{
            return find(nums, left + 1,end, k);
        }
    }

    private void swap(int[] nums, int left, int end){
        int temp = nums[left];
        nums[left] = nums[end];
        nums[end] = temp;
    }
}
```

221. Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

Example:

Input:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Output: 4

```
class Solution {
    public int maximalSquare(char[][] matrix) {
        int m = matrix.length;
        if(m == 0){
            return 0;
        }
        int n = matrix[0].length;
        int[][] dp = new int[m][n];
        int len = 0;
         for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(matrix[i][j] == '1'){
                    if(i > 0 && j > 0){
                        dp[i][j] = Math.min(dp[i-1][j-1], Math.min(dp[i-1][j], dp[i][j-1])
) + 1;
                    }else{
                        dp[i][j] = 1;
                    }
```

```
                }
                len = Math.max(len, dp[i][j]);
            }
        }
        return len * len;
    }
}
```

226. Invert Binary Tree

Invert a binary tree.

Example:

Input:

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

Output:

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

```
public TreeNode invertTree(TreeNode root) {
    if (root == null) {
        return null;
    }
    TreeNode right = invertTree(root.right);
    TreeNode left = invertTree(root.left);
    root.left = right;
    root.right = left;
    return root;
}
```

```
public TreeNode invertTree(TreeNode root) {
    if (root == null) return null;
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode current = queue.poll();
        TreeNode temp = current.left;
        current.left = current.right;
        current.right = temp;
         if (current.left != null) queue.add(current.left);
         if (current.right != null) queue.add(current.right);
    }
     return root;
```

```
}
```

234. Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

Example 1:

Input: 1->2

Output: false

Example 2:

Input: 1->2->2->1

Output: true

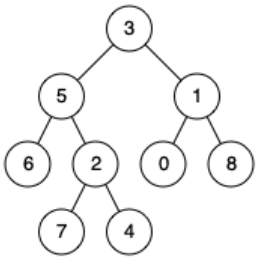
Follow up:

Could you do it in O(n) time and O(1) space?

```
public boolean isPalindrome(ListNode head) {
    ListNode fast = head, slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }
    if (fast != null) { // odd nodes: let right half smaller
        slow = slow.next;
    }
     slow = reverse(slow);
     fast = head;

     while (slow != null) {
         if (fast.val != slow.val) {
             return false;
         }
         fast = fast.next;
         slow = slow.next;
     }
     return true;
}

public ListNode reverse(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode next = head.next;
        head.next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”
Given the following binary tree: root = [3,5,1,6,2,0,8,null,null,7,4]



Example 1:
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes `5` and `1` is `3`.
Example 2:
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

Note:

- All of the nodes' values will be unique.
- p and q are different and both values will exist in the binary tree.

```
class Solution {

    private TreeNode ans;

    public Solution() {
        // Variable to store LCA node.
        this.ans = null;
    }

     private boolean recurseTree(TreeNode currentNode, TreeNode p, TreeNode q) {

         // If reached the end of a branch, return false.
         if (currentNode == null) {
             return false;
         }

         // Left Recursion. If left recursion returns true, set left = 1 else 0
         int left = this.recurseTree(currentNode.left, p, q) ? 1 : 0;

         // Right Recursion
         int right = this.recurseTree(currentNode.right, p, q) ? 1 : 0;

         // If the current node is one of p or q
         int mid = (currentNode == p || currentNode == q) ? 1 : 0;
```

```
25
26
27          // If any two of the flags left, right or mid become True
28          if (mid + left + right >= 2) {
29              this.ans = currentNode;
30          }
31
32          // Return true if any one of the three bool values is True.
33          return (mid + left + right > 0);
34      }
35
36      public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
37          // Traverse the tree
38          this.recurseTree(root, p, q);
39          return this.ans;
40      }
41  }
```

```
1  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2      Map<TreeNode, TreeNode> parent = new HashMap<>();
3      Queue<TreeNode> queue = new LinkedList<>();
4      parent.put(root, null);
5      queue.add(root);
6      while (!parent.containsKey(p) || !parent.containsKey(q)) {
7          TreeNode node = queue.poll();
8          if (node != null) {
9              parent.put(node.left, node);
10             parent.put(node.right, node);
11             queue.add(node.left);
12             queue.add(node.right);
13         }
14     }
15     Set<TreeNode> set = new HashSet<>();
16     while (p != null) {
17         set.add(p);
18         p = parent.get(p);
19     }
20     while (!set.contains(q)) {
21         q = parent.get(q);
22     }
23     return q;
24 }
25
```

238. Product of Array Except Self

Given an array `nums` of *n* integers where *n* > 1, return an array `output`such that `output[i]` is equal to the product of all

the elements of `nums` except `nums[i]`.

Example:

Input: `[1,2,3,4]`

Output: `[24,12,8,6]`

Note: Please solve it **without division** and in O(*n*).

Follow up:

Could you solve it with constant space complexity? (The output array **does not** count as extra space for the purpose of space complexity analysis.)

```
public int[] productExceptSelf(int[] nums) {
    int[] result = new int[nums.length];
    for (int i = 0, tmp = 1; i < nums.length; i++) {
        result[i] = tmp;
        tmp *= nums[i];
    }
    for (int i = nums.length - 1, tmp = 1; i >= 0; i--) {
        result[i] *= tmp;
        tmp *= nums[i];
     }
     return result;
}
```

239. Sliding Window Maximum

Given an array *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:

Input: *nums* = `[1,3,-1,-3,5,3,6,7]`, and *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
```

Note:

You may assume *k* is always valid, 1 ≤ k ≤ input array's size for non-empty array.

Follow up:

Could you solve it in linear time?

We scan the array from 0 to n-1, keep "promising" elements in the deque. The algorithm is amortized O(n) as each element is put and polled once.

At each i, we keep "promising" elements, which are potentially max number in window [i-(k-1),i] or any subsequent window. This means

1. If an element in the deque and it is out of i-(k-1), we discard them. We just need to poll from the head, as we are using a deque and elements are ordered as the sequence in the array
2. Now only those elements within [i-(k-1),i] are in the deque. We then discard elements smaller than a[i] from

the tail. This is because if a[x] <a[i] and x<i, then a[x] has no chance to be the "max" in [i-(k-1),i], or any other subsequent window: a[i] would always be a better candidate.

3. As a result elements in the deque are ordered in both sequence in array and their value. At each step the head of the deque is the max element in [i-(k-1),i]

```
public int[] maxSlidingWindow(int[] a, int k) {
                if (a == null || k <= 0) {
                        return new int[0];
                }
                int n = a.length;
                int[] r = new int[n-k+1];
                int ri = 0;
                // store index
                Deque<Integer> q = new ArrayDeque<>();
                for (int i = 0; i < a.length; i++) {
                        // remove numbers out of range k
                        while (!q.isEmpty() && q.peek() < i - k + 1) {
                                q.poll();
                        }
                        // remove smaller numbers in k range as they are useless
                        while (!q.isEmpty() && a[q.peekLast()] < a[i]) {
                                q.pollLast();
                        }
                        // q contains index... r contains content
                        q.offer(i);
                        if (i >= k - 1) {
                                r[ri++] = a[q.peek()];
                        }
                }
                return r;
        }
```

240. Search a 2D Matrix II

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Example:

Consider the following matrix:

```
[
 [1,  4,  7, 11, 15],
 [2,  5,  8, 12, 19],
 [3,  6,  9, 16, 22],
 [10, 13, 14, 17, 24],
 [18, 21, 23, 26, 30]
]
```

Given target = `5`, return `true`.

Given target = `20`, return `false`

We start search the matrix from top right corner, initialize the current position to top right corner, if the target is greater than the value in current position, then the target can not be in entire row of current position because the row is sorted, if the target is less than the value in current position, then the target can not in the entire column because the column is sorted too. We can rule out one row or one column each time, so the time complexity is O(m+n).

```
public class Solution {
```

```
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length < 1 || matrix[0].length <1) {
            return false;
        }
        int col = matrix[0].length-1;
        int row = 0;
        while(col >= 0 && row <= matrix.length-1) {
            if(target == matrix[row][col]) {
                return true;
            } else if(target < matrix[row][col]) {
                col--;
            } else if(target > matrix[row][col]) {
                row++;
            }
        }
        return false;
    }
}
```

279. Perfect Squares

Given a positive integer *n*, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to *n*.

Example 1:

Input: *n* = `12`

Output: 3

Explanation: `12 = 4 + 4 + 4.`

Example 2:

Input: *n* = `13`

Output: 2

Explanation: `13 = 4 + 9.`

dp[n] indicates that the perfect squares count of the given n, and we have:

```
dp[0] = 0
dp[1] = dp[0]+1 = 1
dp[2] = dp[1]+1 = 2
dp[3] = dp[2]+1 = 3
dp[4] = Min{ dp[4-1*1]+1, dp[4-2*2]+1 }
      = Min{ dp[3]+1, dp[0]+1 }
      = 1
dp[5] = Min{ dp[5-1*1]+1, dp[5-2*2]+1 }
      = Min{ dp[4]+1, dp[1]+1 }
      = 2
                                              .
                                              .
                                              .
dp[13] = Min{ dp[13-1*1]+1, dp[13-2*2]+1, dp[13-3*3]+1 }
       = Min{ dp[12]+1, dp[9]+1, dp[4]+1 }
       = 2
                                              .
                                              .
                                              .
dp[n] = Min{ dp[n - i*i] + 1 },  n - i*i >=0 && i >= 1
```

```
1  public int numSquares(int n) {
```

```
    int[] dp = new int[n + 1];
    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;
    for(int i = 1; i <= n; ++i) {
            int min = Integer.MAX_VALUE;
            int j = 1;
            while(i - j*j >= 0) {
                    min = Math.min(min, dp[i - j*j] + 1);
                    ++j;
            }
            dp[i] = min;
    }
    return dp[n];
}
```

283. Move Zeroes

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Example:

Input: `[0,1,0,3,12]`

Output: `[1,3,12,0,0]`

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

```
public void moveZeroes(int[] nums) {
    if (nums == null || nums.length == 0) return;

    int insertPos = 0;
    for (int num: nums) {
        if (num != 0) nums[insertPos++] = num;
    }

    while (insertPos < nums.length) {
        nums[insertPos++] = 0;
    }
}
```

```
public class Solution {
public void moveZeroes(int[] nums) {

    int j = 0;
    for(int i = 0; i < nums.length; i++) {
        if(nums[i] != 0) {
            int temp = nums[j];
            nums[j] = nums[i];
            nums[i] = temp;
             j++;
         }
```

```
    }
}

}
```

287. Find the Duplicate Number

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Example 1:

Input: `[1,3,4,2,2]`

Output: 2

Example 2:

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

Output: 3

Note:

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than *O*(*n*2).
4. There is only one duplicate number in the array, but it could be repeated more than once.

```java
class Solution {
    public int findDuplicate(int[] nums) {
        Set<Integer> seen = new HashSet<Integer>();
        for (int num : nums) {
            if (seen.contains(num)) {
                return num;
            }
            seen.add(num);
        }

         return -1;
    }
}
```

297. Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Example:

You may serialize the following tree:

```
  1
 / \
2  3
```

```
  /\
 4  5
```

as `"[1,2,3,null,null,4,5]"`

Clarification: The above format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

```
public class Codec {
    private static final String spliter = ",";
    private static final String NN = "X";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append(NN).append(spliter);
        } else {
            sb.append(node.val).append(spliter);
            buildString(node.left, sb);
            buildString(node.right,sb);
        }
    }
    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Deque<String> nodes = new LinkedList<>();
        nodes.addAll(Arrays.asList(data.split(spliter)));
        return buildTree(nodes);
    }

    private TreeNode buildTree(Deque<String> nodes) {
        String val = nodes.remove();
        if (val.equals(NN)) return null;
        else {
            TreeNode node = new TreeNode(Integer.valueOf(val));
            node.left = buildTree(nodes);
            node.right = buildTree(nodes);
            return node;
        }
    }
}

public class Codec {
    public String serialize(TreeNode root) {
        if (root == null) return "";
        Queue<TreeNode> q = new LinkedList<>();
        StringBuilder res = new StringBuilder();
        q.add(root);
        while (!q.isEmpty()) {
            TreeNode node = q.poll();
            if (node == null) {
                res.append("n ");
```

```
                continue;
            }
            res.append(node.val + " ");
            q.add(node.left);
            q.add(node.right);
        }
        return res.toString();
    }

    public TreeNode deserialize(String data) {
        if (data == "") return null;
        Queue<TreeNode> q = new LinkedList<>();
        String[] values = data.split(" ");
        TreeNode root = new TreeNode(Integer.parseInt(values[0]));
        q.add(root);
        for (int i = 1; i < values.length; i++) {
            TreeNode parent = q.poll();
            if (!values[i].equals("n")) {
                TreeNode left = new TreeNode(Integer.parseInt(values[i]));
                parent.left = left;
                q.add(left);
            }
            if (!values[++i].equals("n")) {
                TreeNode right = new TreeNode(Integer.parseInt(values[i]));
                parent.right = right;
                q.add(right);
            }
        }
        return root;
    }
}
```

300. Longest Increasing Subsequence

Given an unsorted array of integers, find the length of longest increasing subsequence.
Example:
Input: `[10,9,2,5,3,7,101,18]`
Output: 4
Explanation: The longest increasing subsequence is `[2,3,7,101]`, therefore the length is `4`.
Note:

- There may be more than one LIS combination, it is only necessary for you to return the length.
- Your algorithm should run in O(*n2*) complexity.

Follow up: Could you improve it to O(*n* log *n*) time complexity?

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int len = 0;

        for(int x : nums) {
            int i = Arrays.binarySearch(dp, 0, len, x);
            if(i < 0) i = -(i + 1);
            dp[i] = x;
            if(i == len) len++;
        }
```

```
        return len;
    }
}

public class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        int len = 0;
        for (int i = 1; i < nums.length; i++) {
            int pos = binarySearch(dp,len,nums[i]);
            if (nums[i] < dp[pos]) dp[pos] = nums[i];
            if (pos > len) {
                len = pos;
                dp[len] = nums[i];
            }
        }
        return len+1;
    }
    private int binarySearch(int[] dp, int len, int val) {
        int left = 0;
        int right = len;
        while(left+1 < right) {
            int mid = left + (right-left)/2;
            if (dp[mid] == val) {
                return mid;
            } else {
                if (dp[mid] < val) {
                    left = mid;
                } else {
                    right = mid;
                }
            }
        }
        if (dp[right] < val) return len+1;
        else if (dp[left] >= val) return left;
        else return right;
    }
}
```

301. Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses `(` and `)`.

Example 1:

Input: `"()())()"`

Output: `["()()()", "(())()"]`

Example 2:

Input: `"(a)())()"`

Output: `["(a)()()", "(a())()"]`

Example 3:

Input: `")("`

Output: [""]

```java
class Solution {
    public List<String> removeInvalidParentheses(String s) {
        Set<String> visited = new HashSet<>();
        List<String> list = new ArrayList<>();
        if(s == null){
            return list;
        }
        Queue<String> queue = new LinkedList<>();
        queue.offer(s);
         visited.add(s);
         // 找到同样长度的，就不需要再分割了
         boolean found = false;
         while(!queue.isEmpty()){
             String str = queue.poll();
             if(isValid(str)){
                 list.add(str);
                 found = true;
             }
             if(found){
                 continue;
             }
             for(int i = 0; i < str.length(); i ++){
                 if(str.charAt(i) != '(' && str.charAt(i) != ')'){
                     continue;
                 }
                 String temp = str.substring(0, i) + str.substring(i + 1) ;
                 if(!visited.contains(temp)){
                     queue.add(temp);
                     visited.add(temp);
                 }

             }
         }
         return list;
     }

     private boolean isValid(String s){
         int count = 0;
         for(int i = 0; i < s.length(); i++){
             char c = s.charAt(i);
             if (c == '(') count++;
             if (c == ')' && count-- == 0) return false;
         }
         return count == 0;
     }
}
```

309. Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the *i*th element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:
Input: [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]

The series of problems are typical dp. The key for dp is to find the variables to represent the states and deduce the transition function.
Of course one may come up with a O(1) space solution directly, but I think it is better to be generous when you think and be greedy when you implement.
The natural states for this problem is the 3 possible transactions : `buy`, `sell`, `rest`. Here `rest` means no transaction on that day (aka cooldown).
Then the transaction sequences can end with any of these three states.
For each of them we make an array, `buy[n]`, `sell[n]` and `rest[n]`.
`buy[i]` means before day `i` what is the maxProfit for any sequence end with `buy`.
`sell[i]` means before day `i` what is the maxProfit for any sequence end with `sell`.
`rest[i]` means before day `i` what is the maxProfit for any sequence end with `rest`.
Then we want to deduce the transition functions for `buy` `sell` and `rest`. By definition we have:

```
buy[i]  = max(rest[i-1]-price, buy[i-1])
sell[i] = max(buy[i-1]+price, sell[i-1])
rest[i] = max(sell[i-1], buy[i-1], rest[i-1])
```

Where `price` is the price of day `i`. All of these are very straightforward. They simply represents :

```
(1) We have to `rest` before we `buy` and
(2) we have to `buy` before we `sell`
```

One tricky point is how do you make sure you `sell` before you `buy`, since from the equations it seems that `[buy, rest, buy]` is entirely possible.
Well, the answer lies within the fact that `buy[i] <= rest[i]` which means `rest[i] = max(sell[i-1], rest[i-1])`. That made sure `[buy, rest, buy]` is never occurred.
A further observation is that and `rest[i] <= sell[i]` is also true therefore

```
rest[i] = sell[i-1]
```

Substitute this in to `buy[i]` we now have 2 functions instead of 3:

```
buy[i] = max(sell[i-2]-price, buy[i-1])
sell[i] = max(buy[i-1]+price, sell[i-1])
```

This is better than 3, but
we can do even better
Since states of day `i` relies only on `i-1` and `i-2` we can reduce the O(n) space to O(1). And here we are at our final solution:
Java

```java
public int maxProfit(int[] prices) {
    int sell = 0, prev_sell = 0, buy = Integer.MIN_VALUE, prev_buy;
    for (int price : prices) {
        prev_buy = buy;
        buy = Math.max(prev_sell - price, prev_buy);
        prev_sell = sell;
        sell = Math.max(prev_buy + price, prev_sell);
    }
    return sell;
}
```

312. Burst Balloons

Given `n` balloons, indexed from `0` to `n-1`. Each balloon is painted with a number on it represented by array `nums`. You are asked to burst all the balloons. If the you burst balloon `i` you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of `i`. After the burst, the `left` and `right` then becomes adjacent.
Find the maximum coins you can collect by bursting the balloons wisely.
Note:

- You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.
- 0 ≤ `n` ≤ 500, 0 ≤ `nums[i]` ≤ 100

Example:
Input: `[3,1,5,8]`
Output: `167`
`Explanation:` nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

```
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] dp = new int[n][n];
    for (int k = 2; k < n; ++k)
        for (int left = 0; left < n - k; ++left) {
            int right = left + k;
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = Math.max(dp[left][right],
                nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][right]);
        }

    return dp[0][n - 1];
}
```

322. Coin Change

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.
Example 1:
Input: coins = `[1, 2, 5]`, amount = `11`
Output: `3`
Explanation: 11 = 5 + 5 + 1
Example 2:
Input: coins = `[2]`, amount = `3`
Output: -1

```
public class Solution {
public int coinChange(int[] coins, int amount) {
    if(amount<1) return 0;
    int[] dp = new int[amount+1];
    int sum = 0;

        while(++sum<=amount) {
                int min = -1;
```

```
        for(int coin : coins) {
                if(sum >= coin && dp[sum-coin]!=-1) {
                        int temp = dp[sum-coin]+1;
                        min = min<0 ? temp : (temp < min ? temp : min);
                }
        }
        dp[sum] = min;
        }
        return dp[amount];
}
}
```

337. House Robber III

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.
Determine the maximum amount of money the thief can rob tonight without alerting the police.
Example 1:
Input: [3,2,3,null,3,null,1]

```
  3
 / \
2  3
 \  \
  3  1
```

Output: 7
Explanation: Maximum amount of money the thief can rob = 3 + 3 + 1 = **7**.
Example 2:
Input: [3,4,5,1,3,null,1]

```
  3
 / \
4  5
/ \  \
1  3  1
```

Output: 9
Explanation: Maximum amount of money the thief can rob = 4 + 5 = **9**.

```
public class Solution {
    public int rob(TreeNode root) {
        int[] num = dfs(root);
        return Math.max(num[0], num[1]);
    }
    private int[] dfs(TreeNode x) {
        if (x == null) return new int[2];
        int[] left = dfs(x.left);
        int[] right = dfs(x.right);
        int[] res = new int[2];
        res[0] = left[1] + right[1] + x.val;
        res[1] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
        return res;
    }
}
```

```java
public int rob(TreeNode root) {
    if (root == null) return 0;
    return Math.max(robInclude(root), robExclude(root));
}

public int robInclude(TreeNode node) {
    if(node == null) return 0;
    return robExclude(node.left) + robExclude(node.right) + node.val;
}

public int robExclude(TreeNode node) {
    if(node == null) return 0;
    return rob(node.left) + rob(node.right);
}

}
```

```java
public int rob(TreeNode root) {
    int[] res = robSub(root);
    return Math.max(res[0], res[1]);
}

private int[] robSub(TreeNode root) {
    if (root == null) return new int[2];

    int[] left = robSub(root.left);
    int[] right = robSub(root.right);
    int[] res = new int[2];

    res[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
    res[1] = root.val + left[0] + right[0];

    return res;
}
```

338. Counting Bits

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example 1:

Input: 2

Output: [0,1,1]

Example 2:

Input: 5

Output: `[0,1,1,2,1,2]`

```java
public int[] countBits(int num) {
    int[] f = new int[num + 1];
    for (int i=1; i<=num; i++) f[i] = f[i >> 1] + (i & 1);
    return f;
}
```

347. Top K Frequent Elements

Given a non-empty array of integers, return the ***k*** most frequent elements.

Example 1:

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

Output: [1,2]

Example 2:

Input: nums = [1], k = 1

Output: [1]

Note:

- You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
- Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.

```
public List<Integer> topKFrequent(int[] nums, int k) {

        List<Integer>[] bucket = new List[nums.length + 1];
        Map<Integer, Integer> frequencyMap = new HashMap<Integer, Integer>();

        for (int n : nums) {
                frequencyMap.put(n, frequencyMap.getOrDefault(n, 0) + 1);
        }

        for (int key : frequencyMap.keySet()) {
                int frequency = frequencyMap.get(key);
                if (bucket[frequency] == null) {
                        bucket[frequency] = new ArrayList<>();
                }
                bucket[frequency].add(key);
        }

        List<Integer> res = new ArrayList<>();

        for (int pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {
                if (bucket[pos] != null) {
                        res.addAll(bucket[pos]);
                }
        }
        return res;
}
```

```
public class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        for(int n: nums){
            map.put(n, map.getOrDefault(n,0)+1);
        }

        // corner case: if there is only one number in nums, we need the bucket has index 1
.
        List<Integer>[] bucket = new List[nums.length+1];
         for(int n:map.keySet()){
            int freq = map.get(n);
            if(bucket[freq]==null)
                bucket[freq] = new LinkedList<>();
```

```
            bucket[freq].add(n);
        }

        List<Integer> res = new LinkedList<>();
        for(int i=bucket.length-1; i>0 && k>0; --i){
            if(bucket[i]!=null){
                List<Integer> list = bucket[i];
                res.addAll(list);
                k-= list.size();
            }
        }

        return res;
    }
}



// use maxHeap. Put entry into maxHeap so we can always poll a number with largest frequency
public class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        for(int n: nums){
            map.put(n, map.getOrDefault(n,0)+1);
        }

        PriorityQueue<Map.Entry<Integer, Integer>> maxHeap =
                        new PriorityQueue<>((a,b)->(b.getValue()-a.getValue()));
        for(Map.Entry<Integer,Integer> entry: map.entrySet()){
            maxHeap.add(entry);
        }

        List<Integer> res = new ArrayList<>();
        while(res.size()<k){
            Map.Entry<Integer, Integer> entry = maxHeap.poll();
            res.add(entry.getKey());
        }
        return res;
    }
}



// use treeMap. Use freqncy as the key so we can get all freqencies in order
public class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
```

```
        for(int n: nums){
            map.put(n, map.getOrDefault(n,0)+1);
        }

        TreeMap<Integer, List<Integer>> freqMap = new TreeMap<>();
        for(int num : map.keySet()){
           int freq = map.get(num);
           if(!freqMap.containsKey(freq)){
               freqMap.put(freq, new LinkedList<>());
           }
           freqMap.get(freq).add(num);
        }

        List<Integer> res = new ArrayList<>();
        while(res.size()<k){
            Map.Entry<Integer, List<Integer>> entry = freqMap.pollLastEntry();
            res.addAll(entry.getValue());
        }
        return res;
    }
}
```

394. Decode String

Given an encoded string, return it's 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; 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 won't be input like `3a` or `2[4]`.

Examples:

`s = "3[a]2[bc]", return "aaabcbc".`

`s = "3[a2[c]]", return "accaccacc".`

`s = "2[abc]3[cd]ef", return "abcabccdcdcdef".`

```
public class Solution {
    public String decodeString(String s) {
        String res = "";
        Stack<Integer> countStack = new Stack<>();
        Stack<String> resStack = new Stack<>();
        int idx = 0;
        while (idx < s.length()) {
            if (Character.isDigit(s.charAt(idx))) {
                int count = 0;
                while (Character.isDigit(s.charAt(idx))) {
                    count = 10 * count + (s.charAt(idx) - '0');
                    idx++;
                }
                countStack.push(count);
            }
            else if (s.charAt(idx) == '[') {
                resStack.push(res);
```

```
                res = "";
                idx++;
            }
            else if (s.charAt(idx) == ']') {
                StringBuilder temp = new StringBuilder (resStack.pop());
                int repeatTimes = countStack.pop();
                for (int i = 0; i < repeatTimes; i++) {
                    temp.append(res);
                }
                res = temp.toString();
                idx++;
            }
            else {
                res += s.charAt(idx++);
            }
        }
        return res;
    }
}

public class Solution {
    private int pos = 0;
    public String decodeString(String s) {
        StringBuilder sb = new StringBuilder();
        String num = "";
        for (int i = pos; i < s.length(); i++) {
            if (s.charAt(i) != '[' && s.charAt(i) != ']' && !Character.isDigit(s.charAt(i))) {
                sb.append(s.charAt(i));
            } else if (Character.isDigit(s.charAt(i))) {
                num += s.charAt(i);
            } else if (s.charAt(i) == '[') {
                pos = i + 1;
                String next = decodeString(s);
                for (int n = Integer.valueOf(num); n > 0; n--) sb.append(next);
                num = "";
                i = pos;
            } else if (s.charAt(i) == ']') {
                pos = i;
                return sb.toString();
            }
        }
        return sb.toString();
    }
}
```

406. Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers `(h, k)`, where `h` is the height of the person and `k` is the number of people in front of this person who have a height greater than or equal to `h`. Write an algorithm to reconstruct the queue.

Note:

The number of people is less than 1,100.

Example

Input:

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

```
 public int[][] reconstructQueue(int[][] people) {
        if (people == null || people.length == 0 || people[0].length == 0)
            return new int[0][0];

        Arrays.sort(people, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                if (b[0] == a[0]) return a[1] - b[1];
                return b[0] - a[0];
            }
        });

        int n = people.length;
        ArrayList<int[]> tmp = new ArrayList<>();
        for (int i = 0; i < n; i++)
            tmp.add(people[i][1], new int[]{people[i][0], people[i][1]});

        int[][] res = new int[people.length][2];
        int i = 0;
        for (int[] k : tmp) {
            res[i][0] = k[0];
            res[i++][1] = k[1];
        }

        return res;
    }
```

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
sort array by the height. (The smaller height has the higher priority. But If people have the same height, for example [5,0] and [5,2], we should consider [5,2] first, because it has more people higher or equal to it, we can treat it is a little shorter than [5,0]).
so after sort, it becomes
[[4,4], [5,2], [5,0], [6,1], [7,1], [7,0]]
(1)[4,4], index = 4,the only position is 4, because it's the shortest--> [][][][][4,4][]
(2)[5,2], index = 2 --> [][][5,2][][4,4][]
(3)[5,0], index = 0, --> [5,0][][5,2][][4,4][]
(4)[6,1], index = 1, the index 1 of remaining unoccupied position ->[5,0][][5,2][6,1][4,4][]
(5)[7,1], index = 1, the index 1 of remaining unoccupied position ->[5,0][][5,2][6,1][4,4][7,1]
(5)[7,0], index = 0, the index 0 of remaining unoccupied position ->[5,0][7,0][5,2][6,1][4,4][7,1]

```
public class Solution {
    public int[][] reconstructQueue(int[][] people) {
        Arrays.sort(people, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                if (a[0] != b[0]) {
                    return a[0] - b[0];
                } else {
                    return b[1] - a[1];
                }
            }
        });
```

```
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < people.length; i++) {
            list.add(i);
        }
        int[][] res = new int[people.length][2];
        for (int i = 0; i < people.length; i++) {
            int index = list.get(people[i][1]);
            res[index][0] = people[i][0];
            res[index][1] = people[i][1];
            list.remove(people[i][1]);
        }
        return res;
    }
}
```

416. Partition Equal Subset Sum

Given a **non-empty** array containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Note:

1. Each of the array element will not exceed 100.
2. The array size will not exceed 200.

Example 1:

Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

Example 2:

Input: [1, 2, 3, 5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

This problem is essentially let us to find whether there are several numbers in a set which are able to sum to a specific value (in this problem, the value is sum/2).

Actually, this is a 0/1 knapsack problem, for each number, we can pick it or not. Let us assume dp[i][j] means whether the specific sum j can be gotten from the first i numbers. If we can pick such a series of numbers from 0-i whose sum is j, dp[i][j] is true, otherwise it is false.

Base case: dp[0][0] is true; (zero number consists of sum 0 is true)

Transition function: For each number, if we don't pick it, dp[i][j] = dp[i-1][j], which means if the first i-1 elements has made it to j, dp[i][j] would also make it to j (we can just ignore nums[i]). If we pick nums[i]. dp[i][j] = dp[i-1][j-nums[i]], which represents that j is composed of the current value nums[i] and the remaining composed of other previous numbers. Thus, the transition function is dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i]]

talking is cheap:

```
public boolean canPartition(int[] nums) {
    int sum = 0;

    for (int num : nums) {
        sum += num;
    }

    if ((sum & 1) == 1) {
```

```
            return false;
        }
        sum /= 2;

        int n = nums.length;
        boolean[][] dp = new boolean[n+1][sum+1];
        for (int i = 0; i < dp.length; i++) {
            Arrays.fill(dp[i], false);
        }

        dp[0][0] = true;

        for (int i = 1; i < n+1; i++) {
            dp[i][0] = true;
        }
        for (int j = 1; j < sum+1; j++) {
            dp[0][j] = false;
        }

        for (int i = 1; i < n+1; i++) {
            for (int j = 1; j < sum+1; j++) {
                dp[i][j] = dp[i-1][j];
                if (j >= nums[i-1]) {
                    dp[i][j] = (dp[i][j] || dp[i-1][j-nums[i-1]]);
                }
            }
        }

        return dp[n][sum];
    }
```

But can we optimize it? It seems that we cannot optimize it in time. But we can optimize in space. We currently use two dimensional array to solve it, but we can only use one dimensional array.

So the code becomes:

```
    public boolean canPartition(int[] nums) {
        int sum = 0;

        for (int num : nums) {
            sum += num;
        }

        if ((sum & 1) == 1) {
            return false;
        }
        sum /= 2;

        int n = nums.length;
        boolean[] dp = new boolean[sum+1];
        Arrays.fill(dp, false);
        dp[0] = true;

        for (int num : nums) {
            for (int i = sum; i > 0; i--) {
                if (i >= num) {
                    dp[i] = dp[i] || dp[i-num];
                }
```

```
            }
        }

        return dp[sum];
    }

    public class Solution {
        public boolean canPartition(int[] nums) {
            // check edge case
            if (nums == null || nums.length == 0) {
                return true;
            }
            // preprocess
            int volumn = 0;
            for (int num : nums) {
                volumn += num;
            }
            if (volumn % 2 != 0) {
                return false;
            }
            volumn /= 2;
            // dp def
            boolean[] dp = new boolean[volumn + 1];
            // dp init
            dp[0] = true;
            // dp transition
            for (int i = 1; i <= nums.length; i++) {
                for (int j = volumn; j >= nums[i-1]; j--) {
                    dp[j] = dp[j] || dp[j - nums[i-1]];
                }
            }
            return dp[volumn];
        }
    }
```

437. Path Sum III

You are given a binary tree in which each node contains an integer value.
Find the number of paths that sum to a given value.
The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).
The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.
Example:
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

```
      10
     /  \
    5   -3
   / \   \
  3   2   11
 / \   \
3  -2   1
```

Return 3. The paths that sum to 8 are:

1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11

```
public class Solution {
    public int pathSum(TreeNode root, int sum) {
        if (root == null) return 0;
        return pathSumFrom(root, sum) + pathSum(root.left, sum) + pathSum(root.right, sum);
    }

    private int pathSumFrom(TreeNode node, int sum) {
        if (node == null) return 0;
        return (node.val == sum ? 1 : 0)
            + pathSumFrom(node.left, sum - node.val) + pathSumFrom(node.right, sum - node.val);
    }
}
```

```
public int pathSum(TreeNode root, int sum) {
        HashMap<Integer, Integer> preSum = new HashMap();
        preSum.put(0,1);
        return helper(root, 0, sum, preSum);
    }

    public int helper(TreeNode root, int currSum, int target, HashMap<Integer, Integer> preSum)
{
        if (root == null) {
            return 0;
        }

        currSum += root.val;
        int res = preSum.getOrDefault(currSum - target, 0);
        preSum.put(currSum, preSum.getOrDefault(currSum, 0) + 1);

        res += helper(root.left, currSum, target, preSum) + helper(root.right, currSum, target,
preSum);
        preSum.put(currSum, preSum.get(currSum) - 1);
        return res;
    }
```

438. Find All Anagrams in a String

Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams in **s**.
Strings consists of lowercase English letters only and the length of both strings **s** and **p** will not be larger than 20,100.
The order of output does not matter.
Example 1:
Input:
s: "cbaebabacd" p: "abc"

Output:
[0, 6]

Explanation:
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
Example 2:

Input:
s: "abab" p: "ab"

Output:
[0, 1, 2]

Explanation:
The substring with start index = 0 is "ab", which is an anagram of "ab".
The substring with start index = 1 is "ba", which is an anagram of "ab".
The substring with start index = 2 is "ab", which is an anagram of "ab".

```
public class Solution {
    public List<Integer> slidingWindowTemplateByHarryChaoyangHe(String s, String t) {
        //init a collection or int value to save the result according the question.
        List<Integer> result = new LinkedList<>();
        if(t.length()> s.length()) return result;

        //create a hashmap to save the Characters of the target substring.
        //(K, V) = (Character, Frequence of the Characters)
        Map<Character, Integer> map = new HashMap<>();
         for(char c : t.toCharArray()){
             map.put(c, map.getOrDefault(c, 0) + 1);
         }
         //maintain a counter to check whether match the target string.
         int counter = map.size();//must be the map size, NOT the string size because the c
har may be duplicate.

         //Two Pointers: begin - left pointer of the window; end - right pointer of the win
dow
         int begin = 0, end = 0;

         //the length of the substring which match the target string.
         int len = Integer.MAX_VALUE;

         //loop at the begining of the source string
         while(end < s.length()){

             char c = s.charAt(end);//get a character

             if( map.containsKey(c) ){
                 map.put(c, map.get(c)-1);// plus or minus one
                 if(map.get(c) == 0) counter--;//modify the counter according the requireme
nt(different condition).
             }
             end++;

             //increase begin pointer to make it invalid/valid again
             while(counter == 0 /* counter condition. different question may have different
 condition */){

                 char tempc = s.charAt(begin);//***be careful here: choose the char at begi
```

```
n pointer, NOT the end pointer
 37                 if(map.containsKey(tempc)){
 38                     map.put(tempc, map.get(tempc) + 1);//plus or minus one
 39                     if(map.get(tempc) > 0) counter++;//modify the counter according the re
quirement(different condition).
 40                 }
 41
 42                 /* save / update(min/max) the result if find a target*/
 43                 // result collections or result int value
 44
 45                 begin++;
 46             }
 47         }
 48         return result;
 49     }
 50 }
```

3) I will give my solution for these questions use the above template one by one

Minimum-window-substring

https://leetcode.com/problems/minimum-window-substring/

```
public class Solution {
    public String minWindow(String s, String t) {
        if(t.length()> s.length()) return "";
        Map<Character, Integer> map = new HashMap<>();
        for(char c : t.toCharArray()){
            map.put(c, map.getOrDefault(c,0) + 1);
        }
        int counter = map.size();

        int begin = 0, end = 0;
        int head = 0;
        int len = Integer.MAX_VALUE;

        while(end < s.length()){
            char c = s.charAt(end);
            if( map.containsKey(c) ){
                map.put(c, map.get(c)-1);
                if(map.get(c) == 0) counter--;
            }
            end++;

            while(counter == 0){
                char tempc = s.charAt(begin);
                if(map.containsKey(tempc)){
                    map.put(tempc, map.get(tempc) + 1);
                    if(map.get(tempc) > 0){
                        counter++;
                    }
                }
                if(end-begin < len){
                    len = end - begin;
                    head = begin;
                }
                begin++;
```

```
            }

        }
        if(len == Integer.MAX_VALUE) return "";
        return s.substring(head, head+len);
    }
}
```

you may find that I only change a little code above to solve the question "Find All Anagrams in a String":

change

```
                if(end-begin < len){
                    len = end - begin;
                    head = begin;
                }
```

to

```
                if(end-begin == t.length()){
                    result.add(begin);
                }
```

longest substring without repeating characters

https://leetcode.com/problems/longest-substring-without-repeating-characters/

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int begin = 0, end = 0, counter = 0, d = 0;

        while (end < s.length()) {
            // > 0 means repeating character
            //if(map[s.charAt(end++)]-- > 0) counter++;
            char c = s.charAt(end);
            map.put(c, map.getOrDefault(c, 0) + 1);
            if(map.get(c) > 1) counter++;
            end++;

            while (counter > 0) {
                //if (map[s.charAt(begin++)]-- > 1) counter--;
                char charTemp = s.charAt(begin);
                if (map.get(charTemp) > 1) counter--;
                map.put(charTemp, map.get(charTemp)-1);
                begin++;
            }
            d = Math.max(d, end - begin);
        }
        return d;
    }
}
```

Longest Substring with At Most Two Distinct Characters

https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/

```
public class Solution {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        Map<Character,Integer> map = new HashMap<>();
        int start = 0, end = 0, counter = 0, len = 0;
        while(end < s.length()){
            char c = s.charAt(end);
            map.put(c, map.getOrDefault(c, 0) + 1);
            if(map.get(c) == 1) counter++;//new char
            end++;
            while(counter > 2){
                char cTemp = s.charAt(start);
```

```
                map.put(cTemp, map.get(cTemp) - 1);
                if(map.get(cTemp) == 0){
                    counter--;
                }
                start++;
            }
            len = Math.max(len, end-start);
        }
        return len;
    }
}
```

Substring with Concatenation of All Words

https://leetcode.com/problems/substring-with-concatenation-of-all-words/

```
public class Solution {
    public List<Integer> findSubstring(String S, String[] L) {
        List<Integer> res = new LinkedList<>();
        if (L.length == 0 || S.length() < L.length * L[0].length())   return res;
        int N = S.length();
        int M = L.length; // *** length
        int wl = L[0].length();
        Map<String, Integer> map = new HashMap<>(), curMap = new HashMap<>();
        for (String s : L) {
            if (map.containsKey(s))   map.put(s, map.get(s) + 1);
            else                      map.put(s, 1);
        }
        String str = null, tmp = null;
        for (int i = 0; i < wl; i++) {
            int count = 0;  // remark: reset count
            int start = i;
            for (int r = i; r + wl <= N; r += wl) {
                str = S.substring(r, r + wl);
                if (map.containsKey(str)) {
                    if (curMap.containsKey(str))   curMap.put(str, curMap.get(str) + 1);
                    else                           curMap.put(str, 1);

                    if (curMap.get(str) <= map.get(str))    count++;
                    while (curMap.get(str) > map.get(str)) {
                        tmp = S.substring(start, start + wl);
                        curMap.put(tmp, curMap.get(tmp) - 1);
                        start += wl;

                        //the same as https://leetcode.com/problems/longest-substring-without-
repeating-characters/
                        if (curMap.get(tmp) < map.get(tmp)) count--;

                    }
                    if (count == M) {
                        res.add(start);
                        tmp = S.substring(start, start + wl);
                        curMap.put(tmp, curMap.get(tmp) - 1);
                        start += wl;
                        count--;
                    }
                }else {
                    curMap.clear();
                    count = 0;
                    start = r + wl;//not contain, so move the start
                }
```

```
            }
            curMap.clear();
        }
        return res;
    }
}
```

Find All Anagrams in a String

https://leetcode.com/problems/find-all-anagrams-in-a-string/

```
public class Solution {
    public List<Integer> findAnagrams(String s, String t) {
        List<Integer> result = new LinkedList<>();
        if(t.length()> s.length()) return result;
        Map<Character, Integer> map = new HashMap<>();
        for(char c : t.toCharArray()){
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        int counter = map.size();

        int begin = 0, end = 0;
        int head = 0;
        int len = Integer.MAX_VALUE;


        while(end < s.length()){
            char c = s.charAt(end);
            if( map.containsKey(c) ){
                map.put(c, map.get(c)-1);
                if(map.get(c) == 0) counter--;
            }
            end++;

            while(counter == 0){
                char tempc = s.charAt(begin);
                if(map.containsKey(tempc)){
                    map.put(tempc, map.get(tempc) + 1);
                    if(map.get(tempc) > 0){
                        counter++;
                    }
                }
                if(end-begin == t.length()){
                    result.add(begin);
                }
                begin++;
            }

        }
        return result;
    }
}
```

448. Find All Numbers Disappeared in an Array

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.
Find all the elements of [1, *n*] inclusive that do not appear in this array.
Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.
Example:
Input:
[4,3,2,7,8,2,3,1]

Output:
[5,6]

```
public List<Integer> findDisappearedNumbers(int[] nums) {
        List<Integer> ret = new ArrayList<Integer>();

        for(int i = 0; i < nums.length; i++) {
            int val = Math.abs(nums[i]) - 1;
            if(nums[val] > 0) {
                nums[val] = -nums[val];
            }
        }

        for(int i = 0; i < nums.length; i++) {
            if(nums[i] > 0) {
                ret.add(i+1);
            }
        }
        return ret;
    }
```

494. Target Sum

You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols `+` and `-`. For each integer, you should choose one from `+` and `-` as its new symbol.
Find out how many ways to assign symbols to make sum of integers equal to target S.
Example 1:
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

`-1+1+1+1+1 = 3`
`+1-1+1+1+1 = 3`
`+1+1-1+1+1 = 3`
`+1+1+1-1+1 = 3`
`+1+1+1+1-1 = 3`

`There are 5 ways to assign symbols to make the sum of nums be target 3.`

```
public class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        if (nums == null || nums.length == 0){
            return 0;
        }
        return helper(nums, 0, 0, S, new HashMap<>());
    }
    private int helper(int[] nums, int index, int sum, int S, Map<String, Integer> map){
        String encodeString = index + "->" + sum;
        if (map.containsKey(encodeString)){
            return map.get(encodeString);
        }
        if (index == nums.length){
            if (sum == S){
                return 1;
            }else {
                return 0;
            }
        }
```

```
        int curNum = nums[index];
        int add = helper(nums, index + 1, sum - curNum, S, map);
        int minus = helper(nums, index + 1, sum + curNum, S, map);
        map.put(encodeString, add + minus);
        return add + minus;
    }
}
```

```
public class Solution {
    int result = 0;

    public int findTargetSumWays(int[] nums, int S) {
        if(nums == null || nums.length == 0) return result;

        int n = nums.length;
        int[] sums = new int[n];
        sums[n - 1] = nums[n - 1];
        for (int i = n - 2; i >= 0; i--)
            sums[i] = sums[i + 1] + nums[i];

        helper(nums, sums, S, 0);
        return result;
    }
    public void helper(int[] nums, int[] sums, int target, int pos){
        if(pos == nums.length){
            if(target == 0) result++;
            return;
        }

        if (sums[pos] < Math.abs(target)) return;

        helper(nums, sums, target + nums[pos], pos + 1);
        helper(nums, sums, target - nums[pos], pos + 1);
    }
}
```

538. Convert BST to Greater Tree

Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

Example:

Input: The root of a Binary Search Tree like this:

```
      5
    /   \
   2    13
```

Output: The root of a Greater Tree like this:

```
     18
    /   \
   20    13
```

```
public class Solution {
    int sum = 0;

    public TreeNode convertBST(TreeNode root) {
        if (root == null) return null;
```

```
        convertBST(root.right);

        root.val += sum;
        sum = root.val;

        convertBST(root.left);

        return root;
    }
}

public class Solution {

    int sum = 0;

    public TreeNode convertBST(TreeNode root) {
        convert(root);
        return root;
    }

    public void convert(TreeNode cur) {
        if (cur == null) return;
        convert(cur.right);
        cur.val += sum;
        sum = cur.val;
        convert(cur.left);
    }

}
```

543. Diameter of Binary Tree

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.

Example:

Given a binary tree

```
      1
     / \
    2  3
   / \
  4  5
```

Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

Note: The length of path between two nodes is represented by the number of edges between them.

```
public class Solution {
    int max = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return max;
    }

    private int maxDepth(TreeNode root) {
        if (root == null) return 0;

        int left = maxDepth(root.left);
        int right = maxDepth(root.right);
```

```
        max = Math.max(max, left + right);

        return Math.max(left, right) + 1;
    }
}

public class Solution {
    public int diameterOfBinaryTree(TreeNode root) {
        if(root == null){
            return 0;
        }
       int dia = depth(root.left) + depth(root.right);
       int ldia = diameterOfBinaryTree(root.left);
       int rdia = diameterOfBinaryTree(root.right);
       return Math.max(dia,Math.max(ldia,rdia));

    }
    public int depth(TreeNode root){
        if(root == null){
            return 0;
        }
        return 1+Math.max(depth(root.left), depth(root.right));
    }

}
```

560. Subarray Sum Equals K

Given an array of integers and an integer **k**, you need to find the total number of continuous subarrays whose sum equals to **k**.

Example 1:
Input:nums = [1,1,1], k = 2
Output: 2

```
public class Solution {
    public int subarraySum(int[] nums, int k) {
        int sum = 0, result = 0;
        Map<Integer, Integer> preSum = new HashMap<>();
        preSum.put(0, 1);

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            if (preSum.containsKey(sum - k)) {
                result += preSum.get(sum - k);
            }
            preSum.put(sum, preSum.getOrDefault(sum, 0) + 1);
        }

        return result;
    }
}
```

572. Subtree of Another Tree

Given two non-empty binary trees **s** and **t**, check whether tree **t** has exactly the same structure and node values with a subtree of **s**. A subtree of **s** is a tree consists of a node in **s** and all of this node's descendants. The tree **s** could also be considered as a subtree of itself.

Example 1:
Given tree s:

```
   3
  / \
 4  5
/ \
1  2
```

Given tree t:

```
 4
/ \
1  2
```

Return **true**, because t has the same structure and node values with a subtree of s.
Example 2:
Given tree s:

```
   3
  / \
 4  5
/ \
1  2
  /
 0
```

Given tree t:

```
 4
/ \
1  2
```

```
public class Solution {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (s == null) return false;
        if (isSame(s, t)) return true;
        return isSubtree(s.left, t) || isSubtree(s.right, t);
    }

    private boolean isSame(TreeNode s, TreeNode t) {
        if (s == null && t == null) return true;
        if (s == null || t == null) return false;

        if (s.val != t.val) return false;

        return isSame(s.left, t.left) && isSame(s.right, t.right);
    }
}
```

```
public class Solution {
 public boolean isSubtree(TreeNode s, TreeNode t) {
        String spreorder = generatepreorderString(s);
        String tpreorder = generatepreorderString(t);

        return spreorder.contains(tpreorder) ;
    }
    public String generatepreorderString(TreeNode s){
        StringBuilder sb = new StringBuilder();
        Stack<TreeNode> stacktree = new Stack();
        stacktree.push(s);
        while(!stacktree.isEmpty()){
           TreeNode popelem = stacktree.pop();
           if(popelem==null)
              sb.append(",#"); // Appending # inorder to handle same values but not subtree
```

```
cases
            else
               sb.append(","+popelem.val);
            if(popelem!=null){
                 stacktree.push(popelem.right);
                 stacktree.push(popelem.left);
            }
        }
        return sb.toString();
    }
}
```

581. Shortest Unsorted Continuous Subarray

Given an integer array, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too.
You need to find the **shortest** such subarray and output its length.
Example 1:
Input: [2, 6, 4, 8, 10, 9, 15]
Output: 5
Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.

```
public int findUnsortedSubarray(int[] A) {
    int n = A.length, beg = -1, end = -2, min = A[n-1], max = A[0];
    for (int i=1;i<n;i++) {
      max = Math.max(max, A[i]);
      min = Math.min(min, A[n-1-i]);
      if (A[i] < max) end = i;
      if (A[n-1-i] > min) beg = n-1-i;
    }
    return end - beg + 1;
}
public class Solution {
    public int findUnsortedSubarray(int[] nums) {
        int n = nums.length;
        int[] temp = nums.clone();
        Arrays.sort(temp);

        int start = 0;
        while (start < n  && nums[start] == temp[start]) start++;

        int end = n - 1;
        while (end > start  && nums[end] == temp[end]) end--;

        return end - start + 1;
    }
}
```

617. Merge Two Binary Trees

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.
You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.
Example 1:
Input:

```
    Tree 1                Tree 2
     1                    2
```

```
    /\          /\
   3  2        1  3
  /            \  \
 5              4  7
```

Output:

Merged tree:

```
     3
    /\
   4  5
  /\  \
 5  4  7
```

```
public class Solution {
    public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
        if (t1 == null && t2 == null) return null;

        int val = (t1 == null ? 0 : t1.val) + (t2 == null ? 0 : t2.val);
        TreeNode newNode = new TreeNode(val);

        newNode.left = mergeTrees(t1 == null ? null : t1.left, t2 == null ? null : t2.left);
        newNode.right = mergeTrees(t1 == null ? null : t1.right, t2 == null ? null : t2.right);

        return newNode;
    }
}
```

```
public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
  if (t1 == null) {
    return t2;
  }

  if (t2 != null) {
    t1.val += t2.val;
    t1.left = mergeTrees(t1.left, t2.left);
    t1.right = mergeTrees(t1.right, t2.right);
  }

  return t1;
}
```

```
public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
  if (t1 == null) {
    return t2;
  }
  // Use stack to help DFS
  Queue<TreeNode[]> queue = new LinkedList<>();
  queue.offer(new TreeNode[] {t1, t2});
  while (!queue.isEmpty()) {
    TreeNode[] cur = queue.poll();
    // no need to merge t2 into t1
    if (cur[1] == null) {
      continue;
    }
    // merge t1 and t2
    cur[0].val += cur[1].val;
    // if node in t1 == null, use node in t2 instead
    // else put both nodes in stack to merge
    if (cur[0].left == null) {
```

```
      cur[0].left = cur[1].left;
    } else {
      queue.offer(new TreeNode[] {cur[0].left, cur[1].left});
    }
    if (cur[0].right == null) {
      cur[0].right = cur[1].right;
    } else {
      queue.offer(new TreeNode[] {cur[0].right, cur[1].right});
    }
  }
  return t1;
}
```

621. Task Scheduler

Given a char array representing tasks CPU need to do. It contains capital letters A to Z where different letters represent different tasks. Tasks could be done without original order. Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.
However, there is a non-negative cooling interval **n** that means between two **same tasks**, there must be at least n intervals that CPU are doing different tasks or just be idle.
You need to return the **least** number of intervals the CPU will take to finish all the given tasks.

Example:
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8
Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.

```
public class Solution {
    public int leastInterval(char[] tasks, int n) {

        int[] c = new int[26];
        for(char t : tasks){
            c[t - 'A']++;
        }
        Arrays.sort(c);
        int i = 25;
        while(i >= 0 && c[i] == c[25]) i--;

        return Math.max(tasks.length, (c[25] - 1) * (n + 1) + 25 - i);
    }
}
```

```
public class Solution {
    public int leastInterval(char[] tasks, int n) {
        if (n == 0) return tasks.length;

        Map<Character, Integer> taskToCount = new HashMap<>();
        for (char c : tasks) {
            taskToCount.put(c, taskToCount.getOrDefault(c, 0) + 1);
        }

        Queue<Integer> queue = new PriorityQueue<>((i1, i2) -> i2 - i1);
        for (char c : taskToCount.keySet()) queue.offer(taskToCount.get(c));

        Map<Integer, Integer> coolDown = new HashMap<>();
        int currTime = 0;
        while (!queue.isEmpty() || !coolDown.isEmpty()) {
            if (coolDown.containsKey(currTime - n - 1)) {
```

```
                queue.offer(coolDown.remove(currTime - n - 1));
            }
            if (!queue.isEmpty()) {
                int left = queue.poll() - 1;
                if (left != 0) coolDown.put(currTime, left);
            }
            currTime++;
        }

        return currTime;
    }
}
```

647. Palindromic Substrings

Given a string, your task is to count how many palindromic substrings in this string.
The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.
Example 1:
Input: "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".

Example 2:
Input: "aaa"
Output: 6
Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

```
public class Solution {
    int count = 0;

    public int countSubstrings(String s) {
        if (s == null || s.length() == 0) return 0;

        for (int i = 0; i < s.length(); i++) { // i is the mid point
            extendPalindrome(s, i, i); // odd length;
            extendPalindrome(s, i, i + 1); // even length
        }

        return count;
    }

    private void extendPalindrome(String s, int left, int right) {
        while (left >=0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            count++; left--; right++;
        }
    }
}
```

```
public int countSubstrings(String s) {
    int n = s.length();
    int res = 0;
    boolean[][] dp = new boolean[n][n];
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i; j < n; j++) {
            dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j - 1]);
            if(dp[i][j]) ++res;
        }
```

```
    }
    return res;
}
```

771. Jewels and Stones

You're given strings `J` representing the types of stones that are jewels, and `S` representing the stones you have. Each character in `S` is a type of stone you have. You want to know how many of the stones you have are also jewels.
The letters in `J` are guaranteed distinct, and all characters in `J` and `S` are letters. Letters are case sensitive, so `"a"` is considered a different type of stone from `"A"`.
Example 1:
Input: J = "aA", S = "aAAbbbb"
Output: 3
Example 2:
Input: J = "z", S = "ZZ"
Output: 0
Note:

- `S` and `J` will consist of letters and have length at most 50.
- The characters in `J` are distinct.

```
public int numJewelsInStones(String J, String S) {
        int count = 0;

        for(int i = 0; i < S.length(); i++) {
            if (J.indexOf(S.charAt(i)) > -1)
                count++;
        }

        return count;
    }
```