344. Reverse String

Write a function that reverses a string. The input string is given as an array of characters *char[]*. Do not allocate extra space for another array. You must do this by modifying the input array in-place with O(1) extra memory.

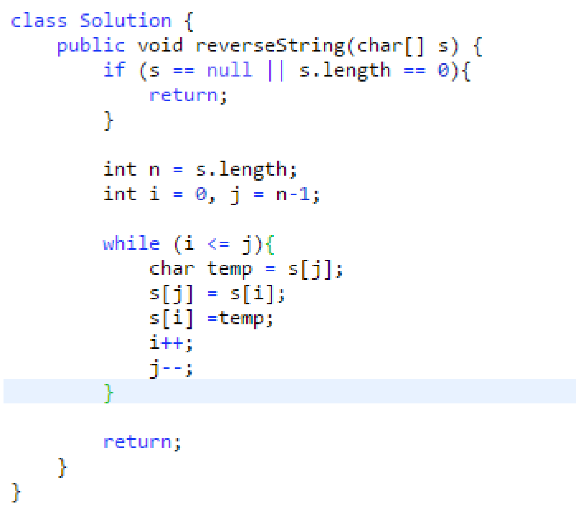
You may assume all the characters consist of printable ascii characters.

Example 1: Input: [“h”, “e”, “l”, “l”, “o”]
Output: [“o”, “l”, “l”, “e”, “h”]

Example 2: Input: [“H”, “a”, “n”, “n”, “a”, “h”]
Output: [“h”, “a”, “n”, “n”, “a”, “H”]

Analysis:
--- A “switch” problem; needs two pointers, *head* and *tail*; needs a temp char, to temporarily store the char to be switched; *head* and *tail* move towards each other simultaneously.

Coding in Java:



```java
class Solution {
    public void reverseString(char[] s) {
        if (s == null || s.length == 0){
            return;
        }

        int n = s.length;
        int i = 0, j = n-1;

        while (i <= j){
            char temp = s[j];
            s[j] = s[i];
            s[i] =temp;
            i++;
            j--;
        }

        return;
    }
}
```

Complexity Analysis:

Time complexity: O(N), where N is the total number of characters in the input string.
Space complexity: O(1) extra space. O(N) space is used where N is the total number of characters in the input string.

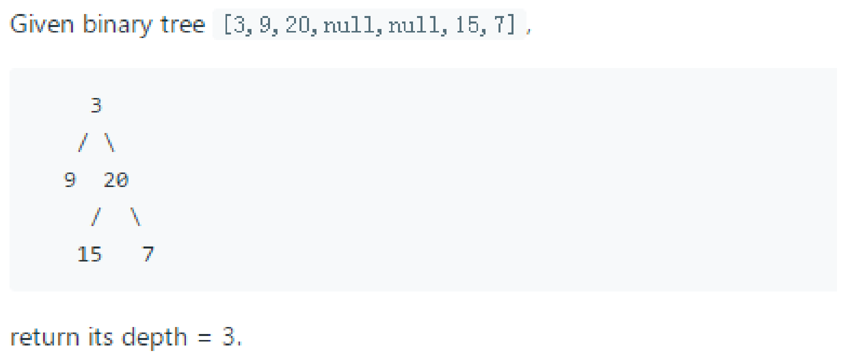
104. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

Example:



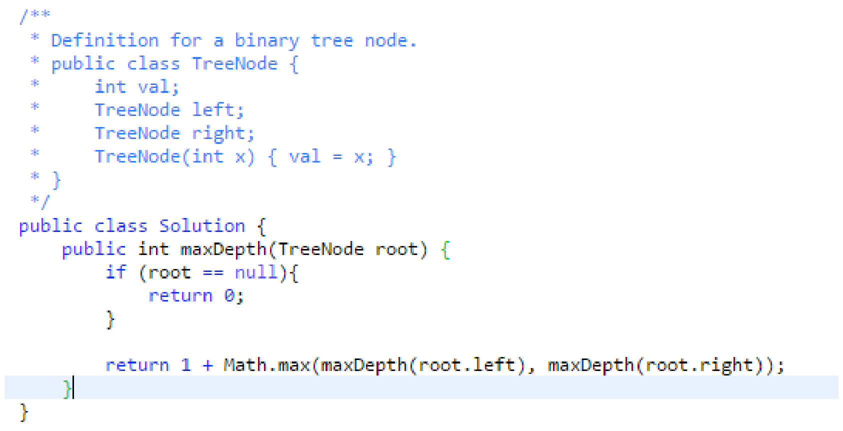
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its depth = 3.

Analysis:
--- Recursive problem; at each node, the height is *1 + Math.max(left_height, right_height)*.

Coding in Java:



```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null){
            return 0;
        }

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

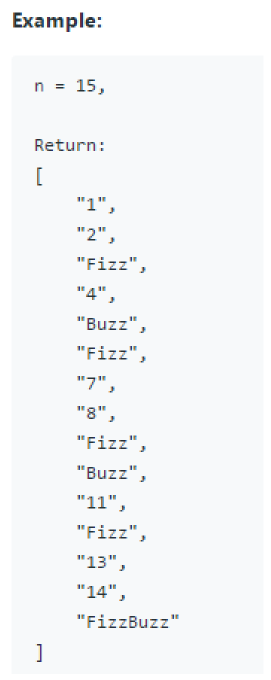
Complexity Analysis:

Time complexity: O(N), where N is the total number of nodes.
Space complexity: No extra memory allocated.

412. Fizz Buzz

Write a program that outputs the string representation of numbers from 1 to n.

But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”. For numbers which are multiples of both three and five output “FizzBuzz”.



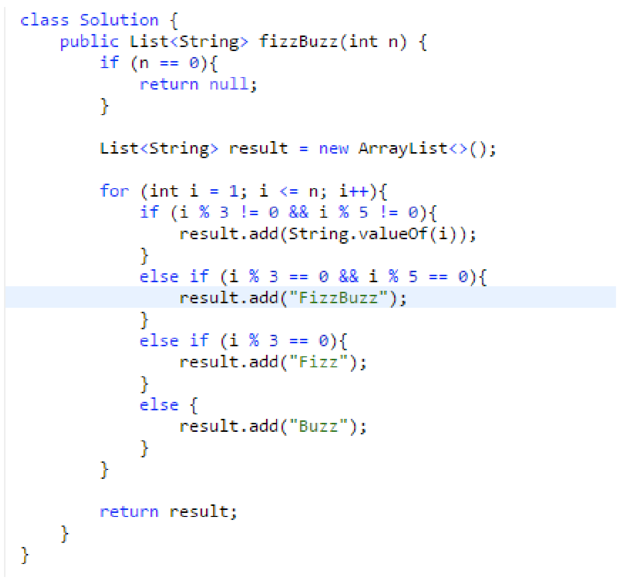
Example:

```
n = 15,

Return:
[
    "1",
    "2",
    "Fizz",
    "4",
    "Buzz",
    "Fizz",
    "7",
    "8",
    "Fizz",
    "Buzz",
    "11",
    "Fizz",
    "13",
    "14",
    "FizzBuzz"
]
```

Analysis:

--- Pretty straightforward. Iterate over 1 to *n*. If *I* is the multiple of 3, output “Fizz”, if multiple of 5, output “Buzz”, if multiple of both 3 and 5, output “FizzBuzz”, if none, then output *i*.

Coding in Java:



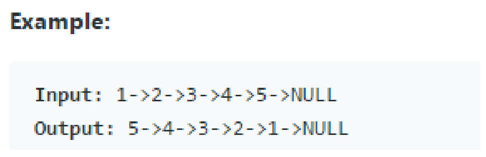
```java
class Solution {
    public List<String> fizzBuzz(int n) {
        if (n == 0){
            return null;
        }

        List<String> result = new ArrayList<>();

        for (int i = 1; i <= n; i++){
            if (i % 3 != 0 && i % 5 != 0){
                result.add(String.valueOf(i));
            }
            else if (i % 3 == 0 && i % 5 == 0){
                result.add("FizzBuzz");
            }
            else if (i % 3 == 0){
                result.add("Fizz");
            }
            else {
                result.add("Buzz");
            }
        }

        return result;
    }
}
```

Complexity analysis:

Time complexity: O(N)
Space complexity: O(1)

206. Reverse Linked List

Reverse a singly linked list.



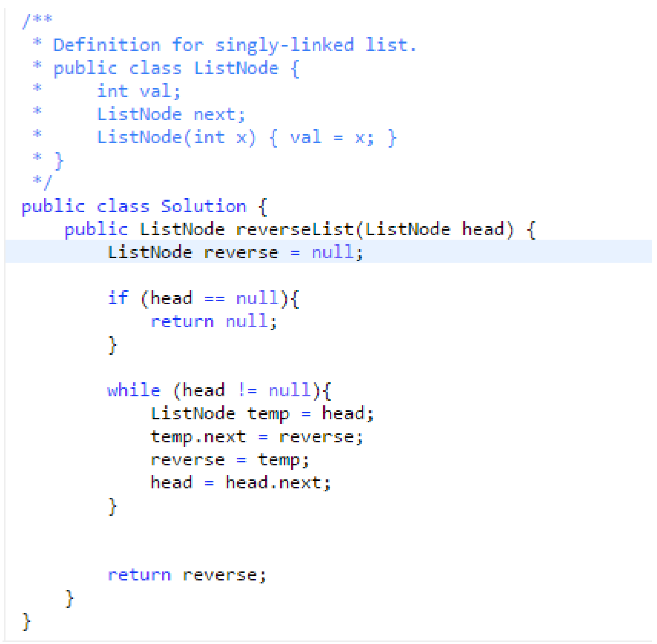
Example:

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

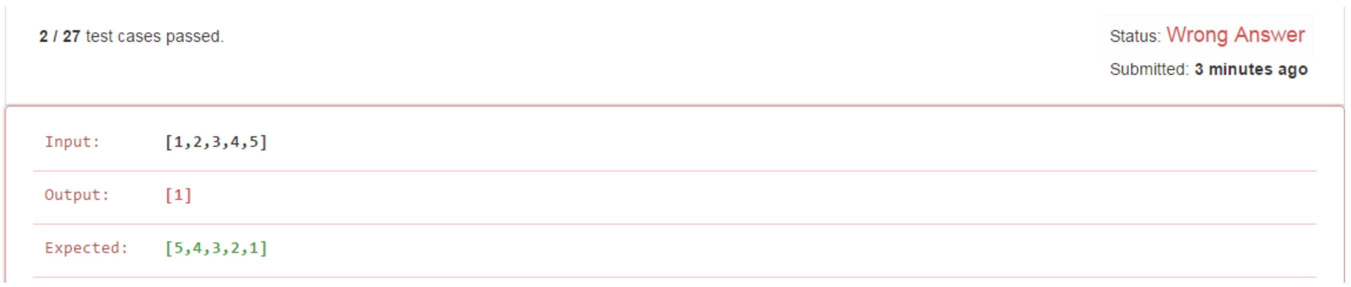
Analysis:
--- Reverse problem; at least two pointers needed, one is “moving pointer”, and another is “recoding pointer”; also, a new Linked List is a must.
--- In each iteration, the “recoding pointer” replicates the location of “moving pointer”, and then connects to the head of new list. Then, the new list’s head goes to the location of “recording pointer”.
--- Pseudo code:
In each iteration:

```
    Temp = head;
    Temp.next = new;
    New = temp;
    Head = head.next;
```

Coding in Java:



```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode reverse = null;

        if (head == null){
            return null;
        }

        while (head != null){
            ListNode temp = head;
            temp.next = reverse;
            reverse = temp;
            head = head.next;
        }


        return reverse;
    }
}
```



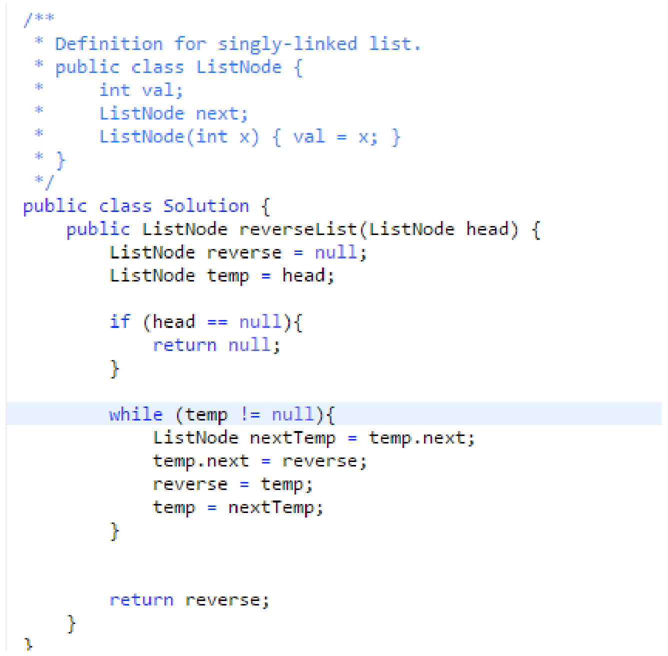
2 / 27 test cases passed.

Status: Wrong Answer

Submitted: **3 minutes ago**

```
Input:      [1,2,3,4,5]

Output:     [1]

Expected:   [5,4,3,2,1]
```

--- Not sure where is wrong???

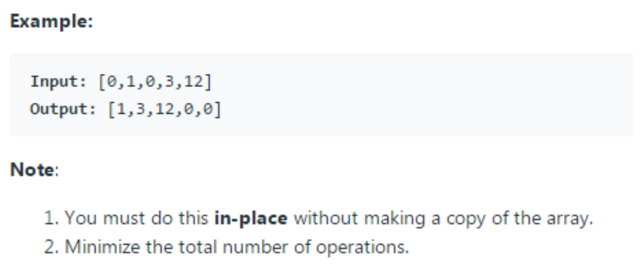


```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode reverse = null;
        ListNode temp = head;

        if (head == null){
            return null;
        }

        while (temp != null){
            ListNode nextTemp = temp.next;
            temp.next = reverse;
            reverse = temp;
            temp = nextTemp;
        }


        return reverse;
    }
}
```

Complexity analysis:
--- Time: O(N)
--- Space: O(1) --- Why?

283. Move Zeros

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.

Analysis:
--- Moving elements in an array; needs two pointers, moving Pointer and anchor. For each element, if 0, *anchor* stands still, and *movingPointer* moves to the next; if not 0, *anchor* and *movingPointer* both move to the next.

Coding in Java:

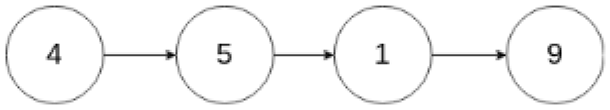
```java
public class Solution {
    public void moveZeroes(int[] nums) {
        if (nums == null || nums.length == 0){
            return;
        }

        int movingPointer = 0;
        int anchor = 0;

        while (movingPointer < nums.length){
            if (nums[movingPointer] != 0){
                nums[anchor] = nums[movingPointer];
                anchor++;
            }

            movingPointer++;
        }

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

        return;
    }
}
```

Complexity analysis
--- Time complexity: O(n)
--- Space complexity: O(1).

237. Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Given linked list --- head = [4, 5, 1, 9], which looks like the following:

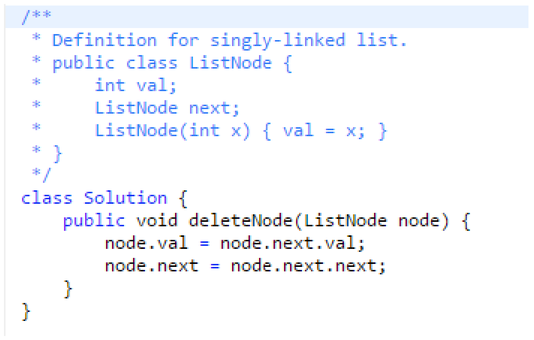


Example 1:

```
Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked
list should become 4 -> 1 -> 9 after calling your function.
```

Analysis:
--- We don’t have access to the *next* node, that means we couldn’t replace the current node with the next node. But we could replace the current *value* with the next *value.*

Coding in Java:



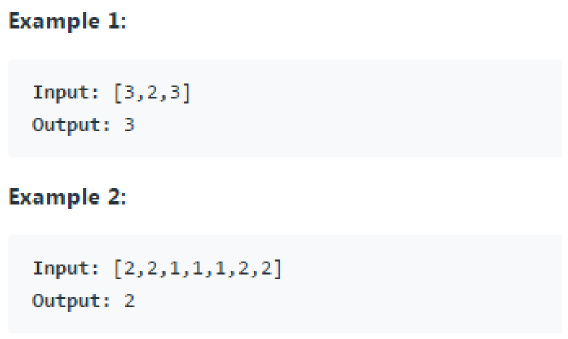
```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public void deleteNode(ListNode node) {
        node.val = node.next.val;
        node.next = node.next.next;
    }
}
```

Complexity analysis:
--- Time complexity: O(1)
--- Space complexity: O(1).

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:

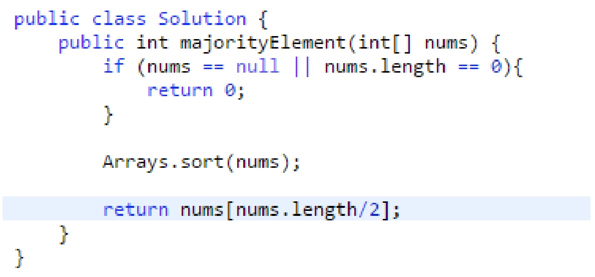
```
Input: [3,2,3]
Output: 3
```

Example 2:

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

Analysis:
--- If "the majority element is the element that appears more than n/2 times", then the element at the *n/2* position must be the "majority".

Coding in Java:

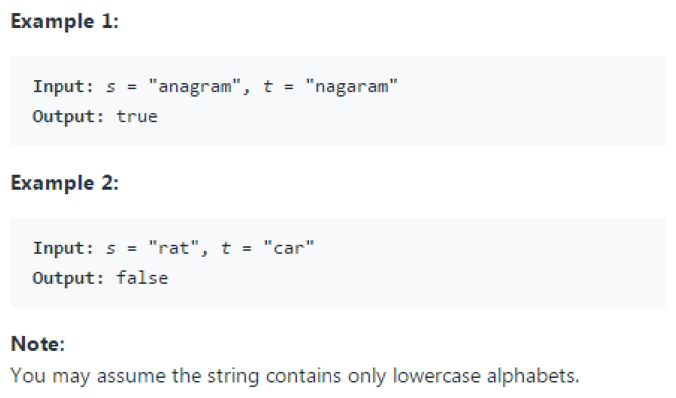


```
public class Solution {
    public int majorityElement(int[] nums) {
        if (nums == null || nums.length == 0){
            return 0;
        }

        Arrays.sort(nums);

        return nums[nums.length/2];
    }
}
```

Complexity analysis:
--- Time complexity: O(nlogn)
--- Space complexity: O(1) or O(n). --- Why??? Need to learn SORTING!

242. Valid Anagram

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.



Example 1:

```
Input: s = "anagram", t = "nagaram"
Output: true
```

Example 2:

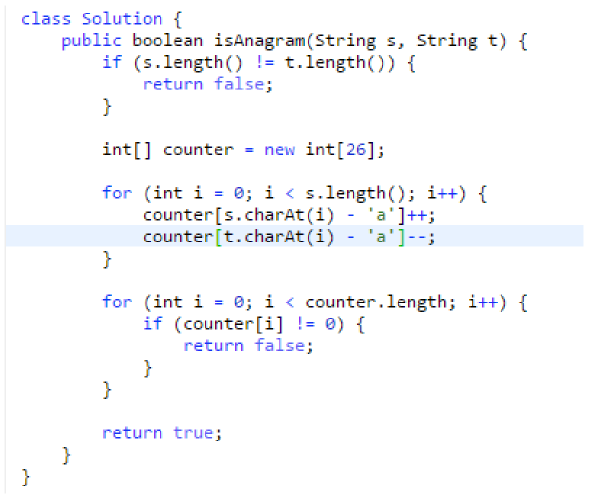
```
Input: s = "rat", t = "car"
Output: false
```

Note:
You may assume the string contains only lowercase alphabets.

Analysis:

--- Typical Hash Table “++” “--” problem.

Coding in Java:



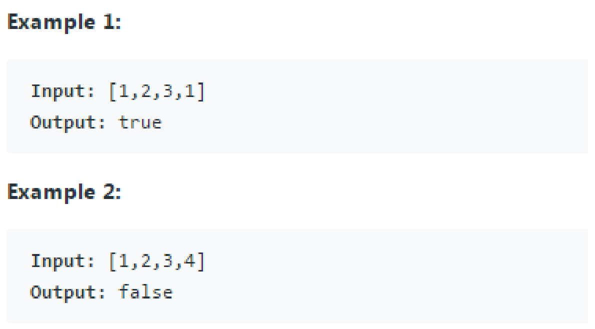
```java
class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }

        int[] counter = new int[26];

        for (int i = 0; i < s.length(); i++) {
            counter[s.charAt(i) - 'a']++;
            counter[t.charAt(i) - 'a']--;
        }

        for (int i = 0; i < counter.length; i++) {
            if (counter[i] != 0) {
                return false;
            }
        }

        return true;
    }
}
```

Complexity analysis:
--- Time complexity: O(n)
--- Space complexity: O(1). The extra table’s size is constant.

217. Contains Duplication

Given an array of integers, find if the array contains any duplicate.

Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.



Example 1:

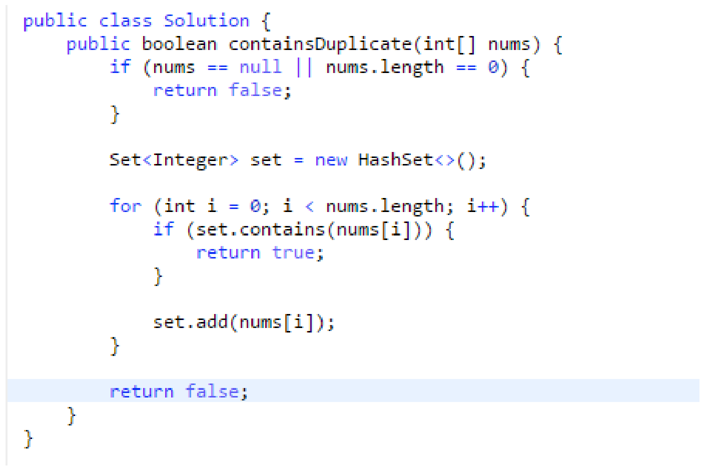
```
Input: [1,2,3,1]
Output: true
```

Example 2:

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

Analysis:
--- Typical Hash Set problem.

Coding in Java:



```java
public class Solution {
    public boolean containsDuplicate(int[] nums) {
        if (nums == null || nums.length == 0) {
            return false;
        }

        Set<Integer> set = new HashSet<>();

        for (int i = 0; i < nums.length; i++) {
            if (set.contains(nums[i])) {
                return true;
            }

            set.add(nums[i]);
        }

        return false;
    }
}
```

Complexity analysis:
--- Time complexity: O(n)
--- Space complexity: O(n), linear with the number of elements in it.

122. Best Time to Buy and Sell Stock II

Say you have an array for which the *ith* element is the price of a given stock on day *i*.

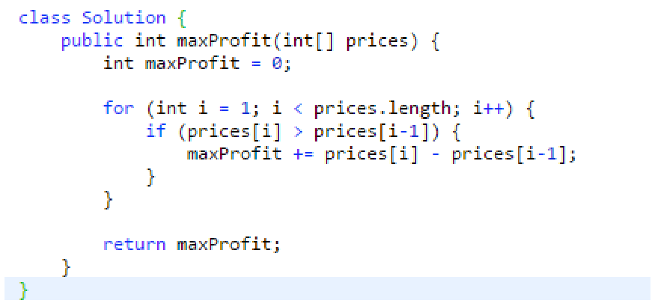
Design an algorithm to find the maximum profit. You may complete as many transactions as you like.

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Analysis

--- Strategy: Do not want to miss any “Peak” right after any “Valley”.

Coding in Java:

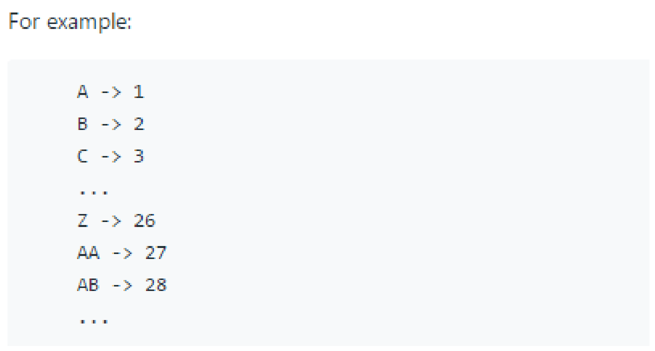


```java
class Solution {
    public int maxProfit(int[] prices) {
        int maxProfit = 0;

        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i-1]) {
                maxProfit += prices[i] - prices[i-1];
            }
        }

        return maxProfit;
    }
}
```

Complexity analysis:
--- Time complexity: O(n)
--- Space complexity: O(1).

171. Excel Sheet Column Number

Given a column title as appear in an Excel sheet, return its corresponding column number.



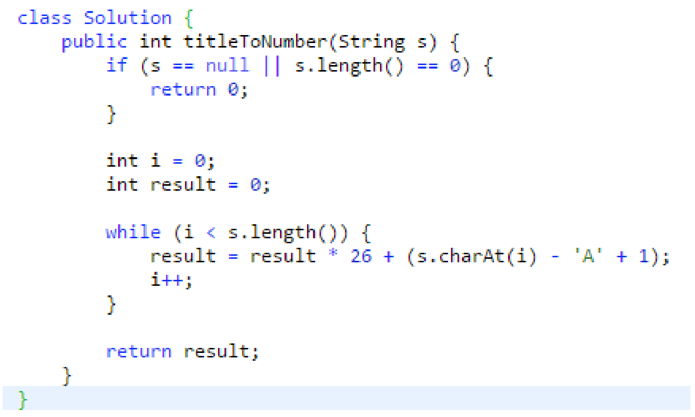
For example:

```
    A -> 1
    B -> 2
    C -> 3
    ...
    Z -> 26
    AA -> 27
    AB -> 28
    ...
```

Analysis
--- A = 1, B = 2, C = 3, …, Z = 26
--- AA = 26 * 1 + 1, AB = 26 * 1 + 1, …, AZ = 26 * 1 + 26, BA = 26 * 2 + 1, BB = 26 * 2 + 2, BZ = 26 * 2 + 26

--- CA = 26 * 3 + 1, CB = 26 * 3 + 2, …, CZ = 26 * 3 + 26
……
--- ZA = 26 * 26 + 1, ZB = 26 * 26 + 2, …, ZZ = 26 * 26 + 26
--- AAA = 26 * 26 + 26 * 1 + 1…

Coding in Java:

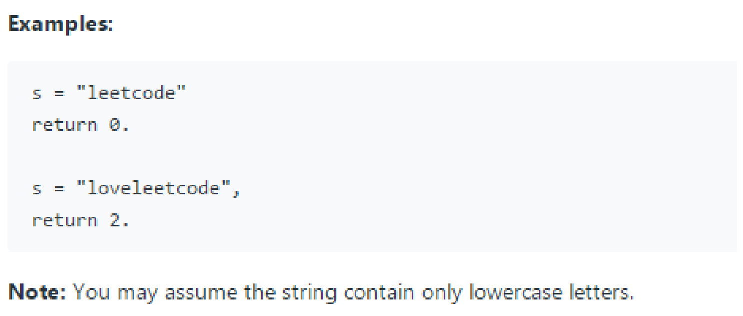


```java
class Solution {
    public int titleToNumber(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        int i = 0;
        int result = 0;

        while (i < s.length()) {
            result = result * 26 + (s.charAt(i) - 'A' + 1);
            i++;
        }

        return result;
    }
}
```

Complexity analysis:
--- Time complexity: ???
--- Space complexity: ???

387. First Unique Character in a String

Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1.



Examples:

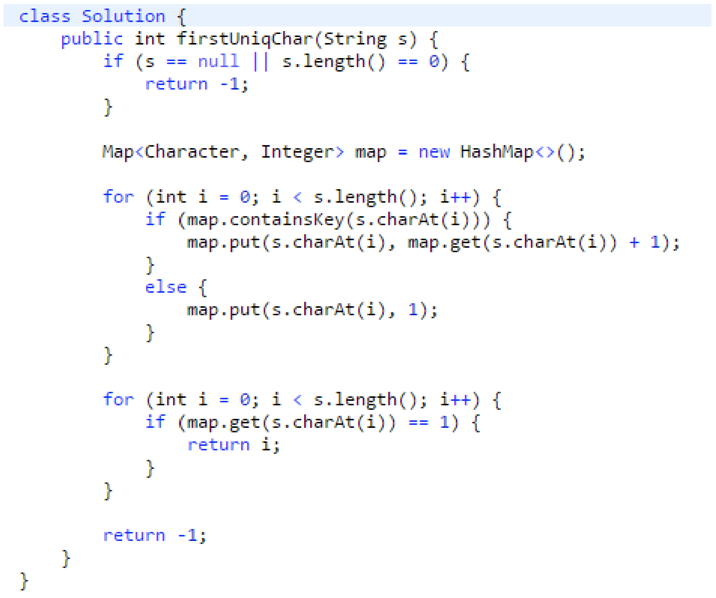
```
s = "leetcode"
return 0.

s = "loveleetcode",
return 2.
```

Note: You may assume the string contain only lowercase letters.

Analysis:

--- Typical Hash Map problem.

Coding in Java:



```
class Solution {
    public int firstUniqChar(String s) {
        if (s == null || s.length() == 0) {
            return -1;
        }

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

        for (int i = 0; i < s.length(); i++) {
            if (map.containsKey(s.charAt(i))) {
                map.put(s.charAt(i), map.get(s.charAt(i)) + 1);
            }
            else {
                map.put(s.charAt(i), 1);
            }
        }

        for (int i = 0; i < s.length(); i++) {
            if (map.get(s.charAt(i)) == 1) {
                return i;
            }
        }

        return -1;
    }
}
```

Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(N), since we have to keep a hash map with N elements.

268. Missing Number

Given an array containing *n* distinct numbers taken from 0, 1, 2, …, n, find the one that is missing from the array.

Example 1:

```
Input: [3,0,1]
Output: 2
```

Example 2:

```
Input: [9,6,4,2,3,5,7,0,1]
Output: 8
```

Analysis:

--- Pretty obvious, use sort.

Coding in Java:

```
public class Solution {
    public int missingNumber(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int n = nums.length;

        Arrays.sort(nums);

        if (nums[0] != 0) {
            return 0;
        }

        if (nums[n-1] != n) {
            return n;
        }

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

        return -1;
    }
}
```

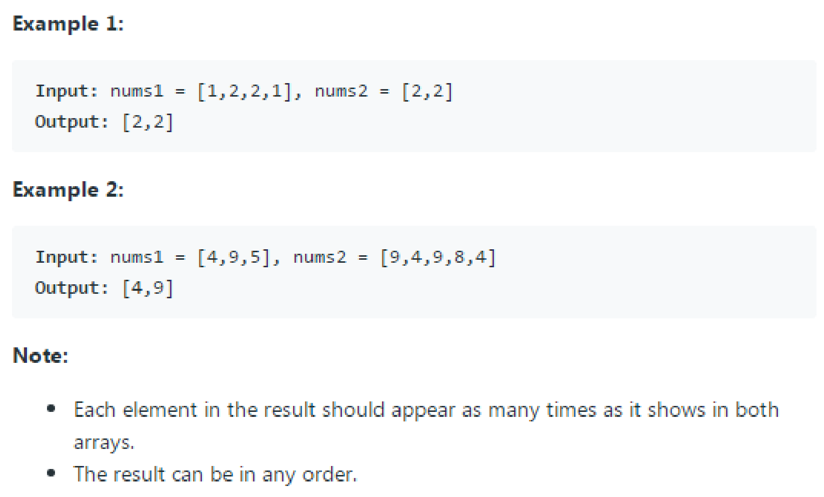
Complexity analysis:

--- Time complexity: O(nlogn)

--- Space complexity: O(1).

350. Intersection of Two Arrays II

Given two arrays, write a function to compute their intersection.



Example 1:

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2,2]
```

Example 2:

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [4,9]
```

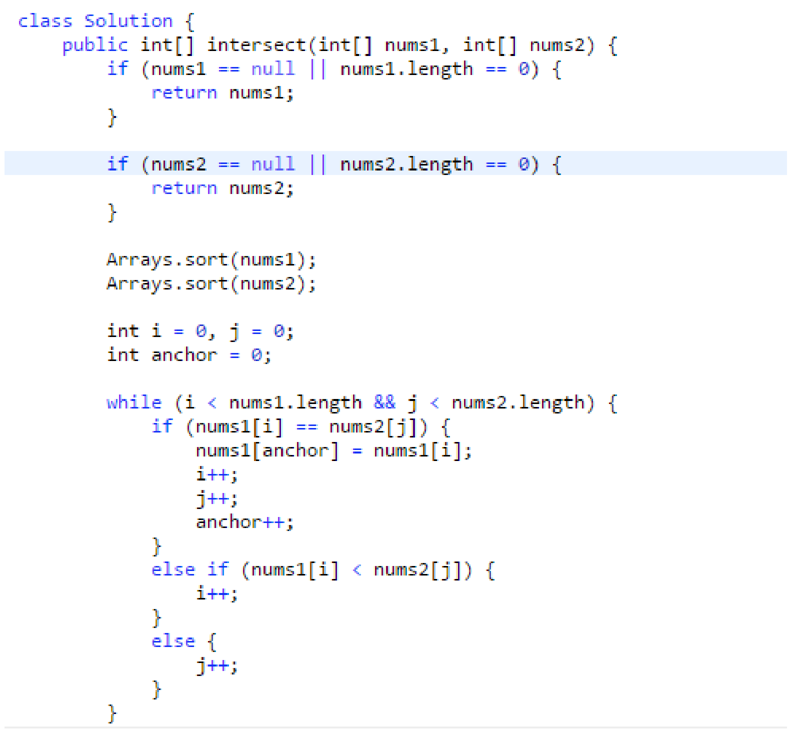
Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Analysis:

--- Typical: first sort, then 3 pointers, with 2 movers and 1 anchor.

Coding in Java:



```
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        if (nums1 == null || nums1.length == 0) {
            return nums1;
        }

        if (nums2 == null || nums2.length == 0) {
            return nums2;
        }

        Arrays.sort(nums1);
        Arrays.sort(nums2);

        int i = 0, j = 0;
        int anchor = 0;

        while (i < nums1.length && j < nums2.length) {
            if (nums1[i] == nums2[j]) {
                nums1[anchor] = nums1[i];
                i++;
                j++;
                anchor++;
            }
            else if (nums1[i] < nums2[j]) {
                i++;
            }
            else {
                j++;
            }
        }
        int[] result = new int[anchor];

        for (int k = 0; k < anchor; k++) {
            result[k] = nums1[k];
        }

        return result;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(nlogn).

21. Merge Two Sorted Lists

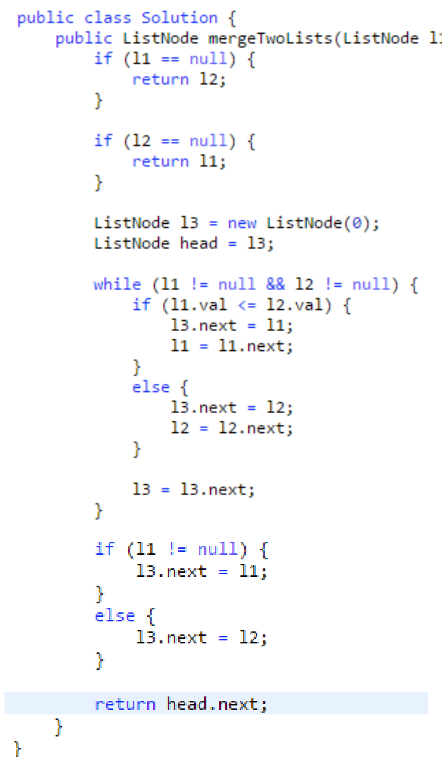
Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Example:

```
Input: 1->2->4, 1->3->4
Output: 1->1->2->3->4->4
```

Analysis:
--- Typical: two pointers, fast and slow, and 1 anchor.

Coding in Java:



```
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }

        if (l2 == null) {
            return l1;
        }

        ListNode l3 = new ListNode(0);
        ListNode head = l3;

        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                l3.next = l1;
                l1 = l1.next;
            }
            else {
                l3.next = l2;
                l2 = l2.next;
            }

            l3 = l3.next;
        }

        if (l1 != null) {
            l3.next = l1;
        }
        else {
            l3.next = l2;
        }

        return head.next;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(N)

121. Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

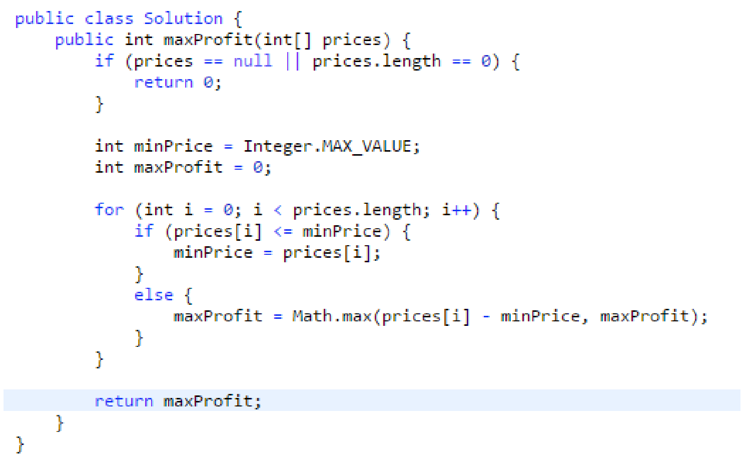
If you were only permitted to complete at most one transaction, design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Analysis:
--- For each day, check if the price is lower than the lowest prices. If lower, then not sell, but only adjust the lower price; if higher, then calculate the profit, compare it with the maximum

profit so far. If higher, then sell, and adjust the maximum profit; if lower, then not sell, not do any adjustment, just skip.

Coding in Java:



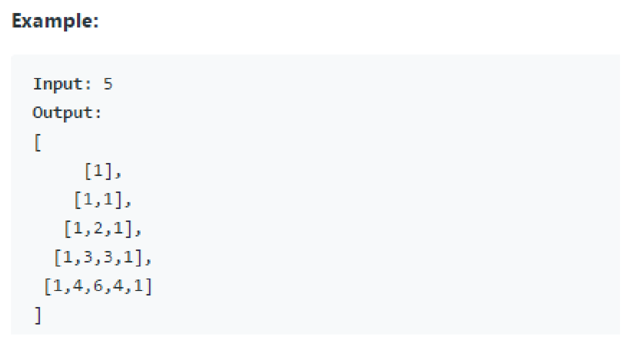
```java
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }

        int minPrice = Integer.MAX_VALUE;
        int maxProfit = 0;

        for (int i = 0; i < prices.length; i++) {
            if (prices[i] <= minPrice) {
                minPrice = prices[i];
            }
            else {
                maxProfit = Math.max(prices[i] - minPrice, maxProfit);
            }
        }

        return maxProfit;
    }
}
```

Complexity analysis:

--- Time complexity: O(N)
--- Space complexity: O(1). Only two variables are used.

118. Pascal’s Triangle

Given a non-negative integer *numRows*, generate the first *numRows* of Pascal’s triangle.



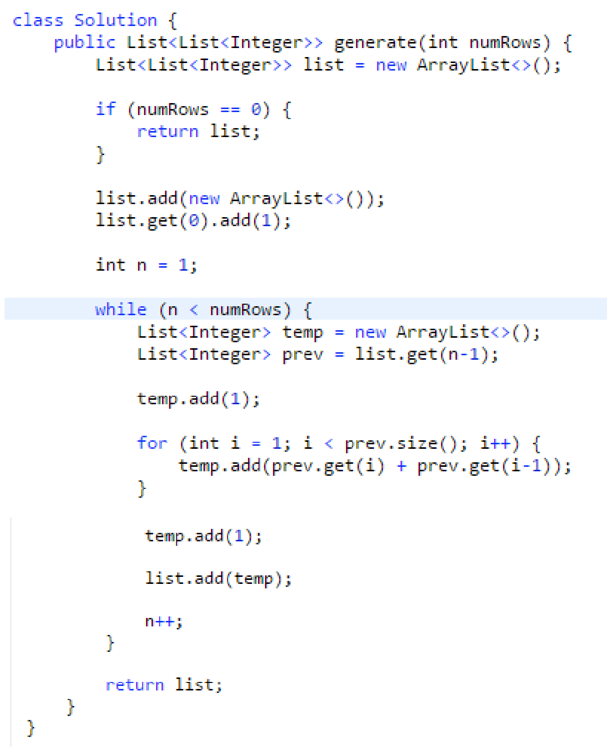
Example:

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

Analysis:

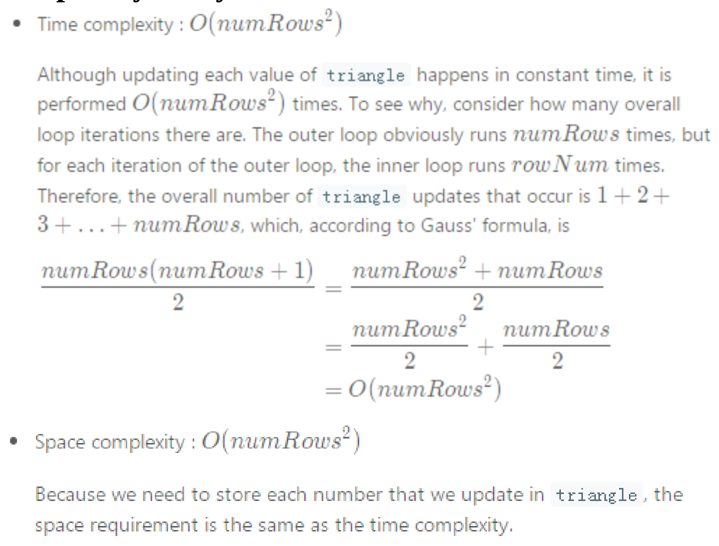
--- The number of inner lists in the outer list is *numRows*; for each inner list, the first/last element is always 1; the *ith* element is the sum of the *ith* and *(i-1)th* elements of the previous inner list.

Coding in Java:



```
class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> list = new ArrayList<>();

        if (numRows == 0) {
            return list;
        }

        list.add(new ArrayList<>());
        list.get(0).add(1);

        int n = 1;

        while (n < numRows) {
            List<Integer> temp = new ArrayList<>();
            List<Integer> prev = list.get(n-1);

            temp.add(1);

            for (int i = 1; i < prev.size(); i++) {
                temp.add(prev.get(i) + prev.get(i-1));
            }

             temp.add(1);

             list.add(temp);

             n++;
         }

         return list;
     }
 }
```

Complexity analysis:



- Time complexity : $O(numRows^2)$

  Although updating each value of `triangle` happens in constant time, it is performed $O(numRows^2)$ times. To see why, consider how many overall loop iterations there are. The outer loop obviously runs $numRows$ times, but for each iteration of the outer loop, the inner loop runs $rowNum$ times. Therefore, the overall number of `triangle` updates that occur is $1 + 2 + 3 + \ldots + numRows$, which, according to Gauss' formula, is

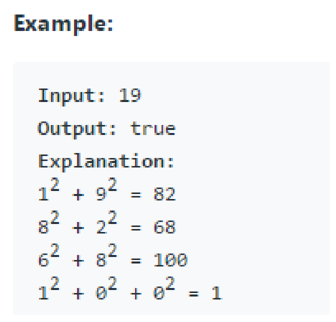
$$\begin{aligned} \frac{numRows(numRows+1)}{2} &= \frac{numRows^2 + numRows}{2} \\ &= \frac{numRows^2}{2} + \frac{numRows}{2} \\ &= O(numRows^2) \end{aligned}$$

- Space complexity : $O(numRows^2)$

  Because we need to store each number that we update in `triangle`, the space requirement is the same as the time complexity.

202. Happy Number

Write an algorithm to determine if a number is “happy”.



Example:

```
Input: 19
Output: true
Explanation:
```
$1^2 + 9^2 = 82$

$8^2 + 2^2 = 68$

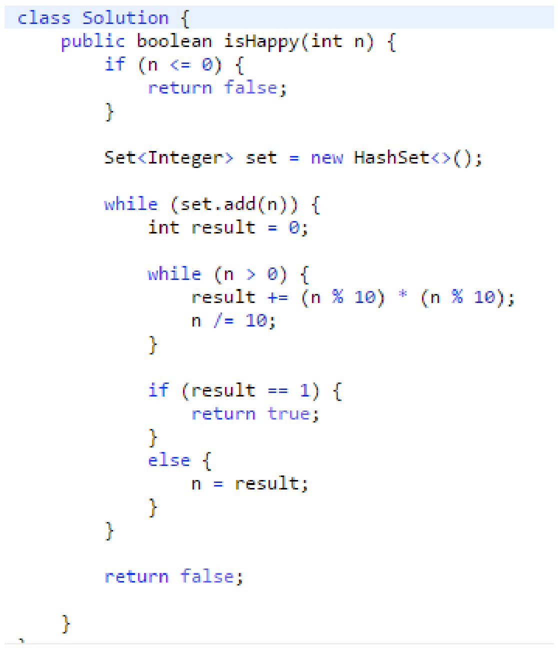
$6^2 + 8^2 = 100$

$1^2 + 0^2 + 0^2 = 1$

Analysis:

--- Once the current sum cannot be added to set, return false. (Don’t quite understand)

Coding in Java:



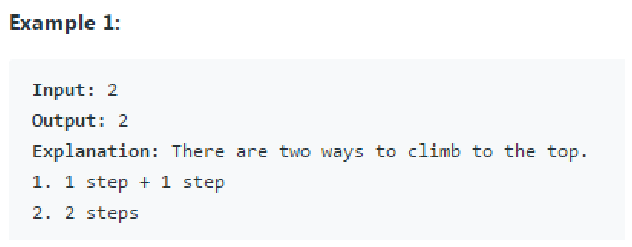
```java
class Solution {
    public boolean isHappy(int n) {
        if (n <= 0) {
            return false;
        }

        Set<Integer> set = new HashSet<>();

        while (set.add(n)) {
            int result = 0;

            while (n > 0) {
                result += (n % 10) * (n % 10);
                n /= 10;
            }

            if (result == 1) {
                return true;
            }
            else {
                n = result;
            }
        }

        return false;

    }
```

Complexity analysis:
--- Time complexity:
--- Space complexity: O(1)

70. Climbing Stairs

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

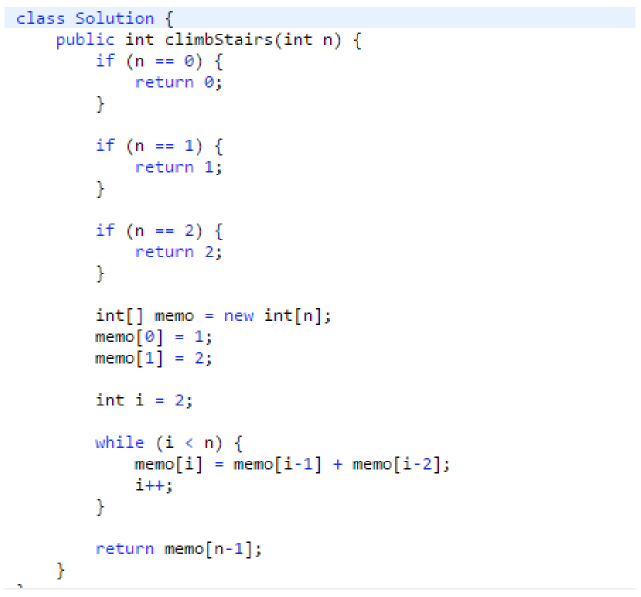


Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Analysis:
--- At the *ith* step, the ways to reach to the *ith* step depends on (ways to *(i-1)th* + ways to *(i-2)th*).

Coding in Java:



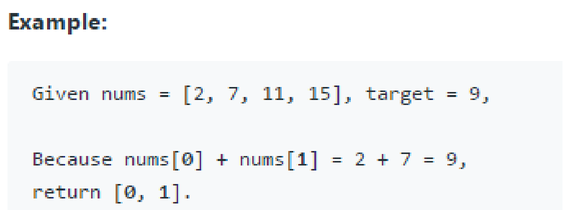
```
class Solution {
    public int climbStairs(int n) {
        if (n == 0) {
            return 0;
        }

        if (n == 1) {
            return 1;
        }

        if (n == 2) {
            return 2;
        }

        int[] memo = new int[n];
        memo[0] = 1;
        memo[1] = 2;

        int i = 2;

        while (i < n) {
            memo[i] = memo[i-1] + memo[i-2];
            i++;
        }

        return memo[n-1];
    }
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(N)

1. Two Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

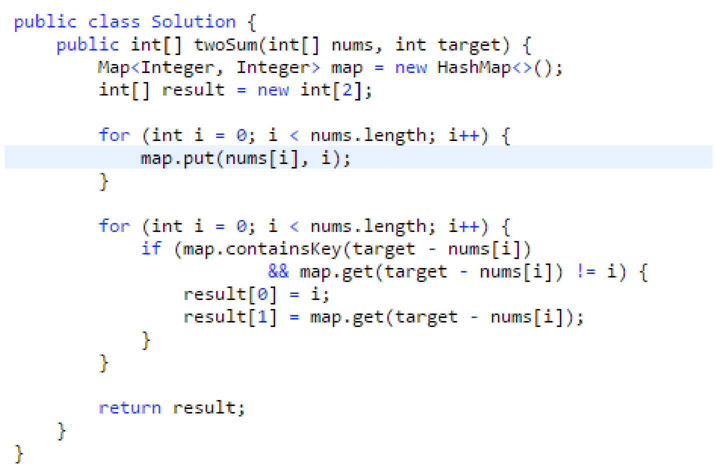


Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

Analysis:
--- Typical hash map problem.

Coding in Java:

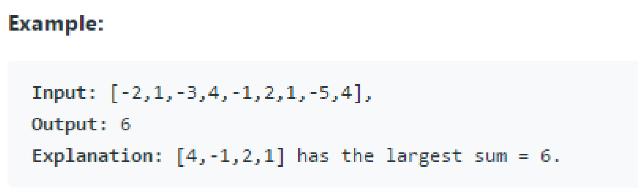


```java
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        int[] result = new int[2];

        for (int i = 0; i < nums.length; i++) {
            map.put(nums[i], i);
        }

        for (int i = 0; i < nums.length; i++) {
            if (map.containsKey(target - nums[i])
                    && map.get(target - nums[i]) != i) {
                result[0] = i;
                result[1] = map.get(target - nums[i]);
            }
        }

        return result;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(N).

53. Maximum Subarray

Given an integer array *nums*, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

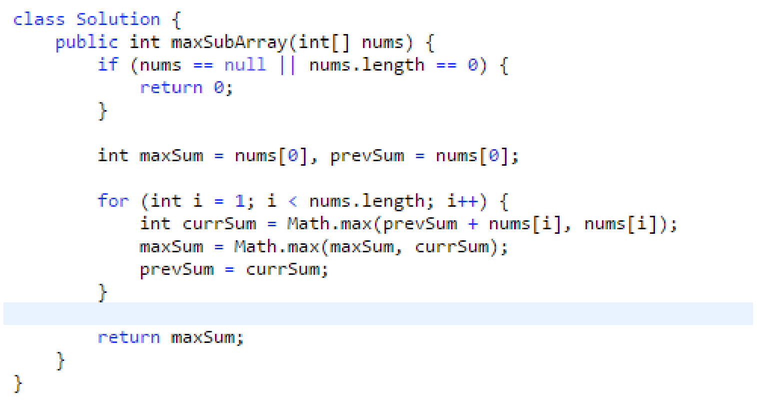


Example:

```
Input: [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Analysis:
--- At each element, we have 3 questions: 1) Should we add it to the previous sum? 2) Should we start over with this element? 3) Should we stop right here?   For 3), the answer is no. Because if we stop right here, we'd never know if there'll be larger sum afterwards. So, we should not stop before reaching the end. But while we are moving, we should keep the largest sum. Once the new sum is larger, we adjust this largest sum in a timely manner.

Coding in Java:

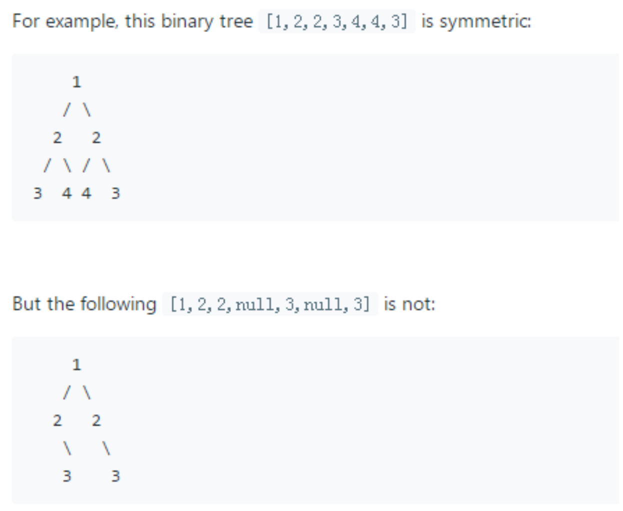


```
class Solution {
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int maxSum = nums[0], prevSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            int currSum = Math.max(prevSum + nums[i], nums[i]);
            maxSum = Math.max(maxSum, currSum);
            prevSum = currSum;
        }

        return maxSum;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(1).

101. Symmetric Tree

Given a binary tree, check whether it is a mirror of itself.



For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

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

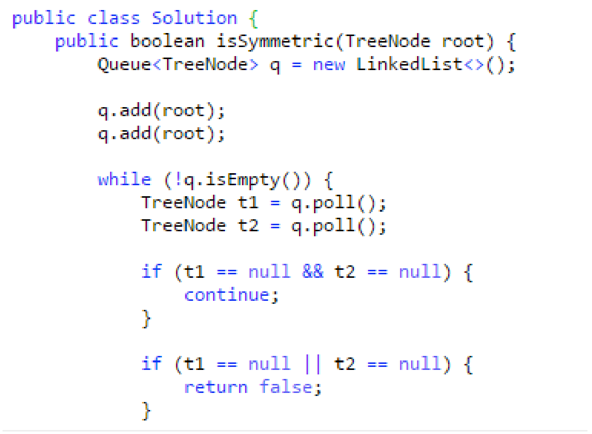
But the following `[1,2,2,null,3,null,3]` is not:

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

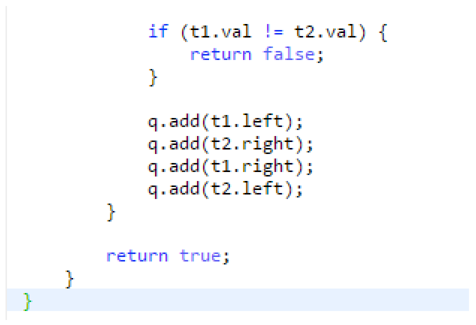
Analysis:
--- At each depth, we have two nodes: left and right. If 1) left.value = right.value, and 2) left.left.value = right.right.value, and 3) left.right.value = right.left.value, then symmetric.

Coding in Java:

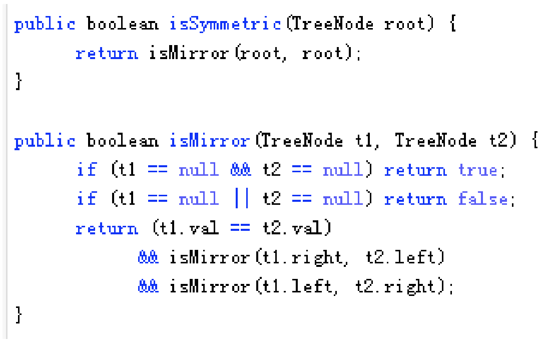
--- Iterative:



```
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        Queue<TreeNode> q = new LinkedList<>();

        q.add(root);
        q.add(root);

        while (!q.isEmpty()) {
            TreeNode t1 = q.poll();
            TreeNode t2 = q.poll();

            if (t1 == null && t2 == null) {
                continue;
            }

            if (t1 == null || t2 == null) {
                return false;
            }
```



```
            if (t1.val != t2.val) {
                return false;
            }

            q.add(t1.left);
            q.add(t2.right);
            q.add(t1.right);
            q.add(t2.left);
        }

        return true;
    }
}
```



```
public boolean isSymmetric(TreeNode root) {
    return isMirror(root, root);
}

public boolean isMirror(TreeNode t1, TreeNode t2) {
    if (t1 == null && t2 == null) return true;
    if (t1 == null || t2 == null) return false;
    return (t1.val == t2.val)
        && isMirror(t1.right, t2.left)
        && isMirror(t1.left, t2.right);
}
```

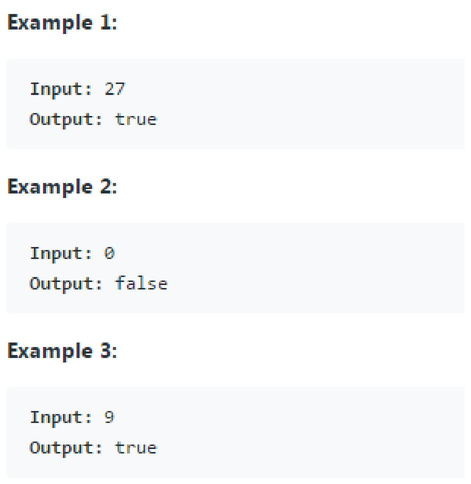
Complexity analysis:

--- Iterative: Time complexity – O(N), Space complexity – O(N)

--- Recursive: Time complexity – O(N), Space complexity – O(N).

326. Power of Three

Given an integer, write a function to determine if it is a power of three.



Example 1:

```
Input: 27
Output: true
```

Example 2:

```
Input: 0
Output: false
```

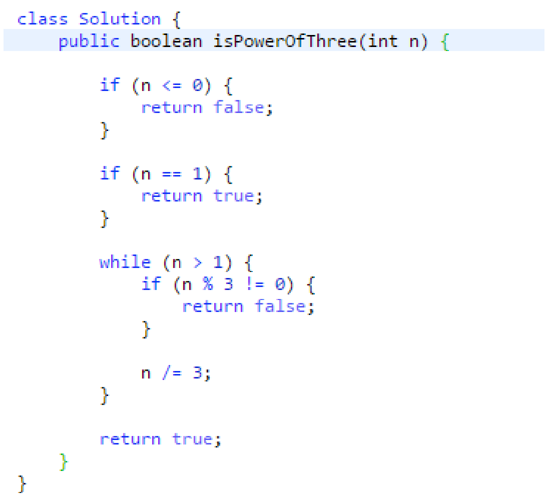
Example 3:

```
Input: 9
Output: true
```

Analysis:

--- Pretty obvious, keep dividing by 3, until n = 1.

Coding in Java:



```java
class Solution {
    public boolean isPowerOfThree(int n) {

        if (n <= 0) {
            return false;
        }

        if (n == 1) {
            return true;
        }

        while (n > 1) {
            if (n % 3 != 0) {
                return false;
            }

            n /= 3;
        }

        return true;
    }
}
```

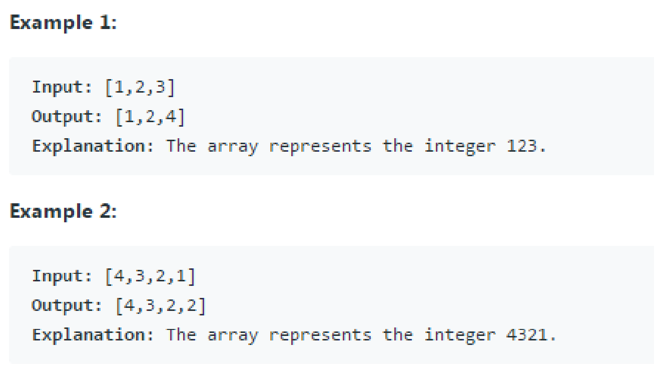
Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(1)

66. Plus One

Given a non-empty array of digits representing a non-negative integer, plus one to the integer.



Example 1:

```
Input: [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
```

Example 2:

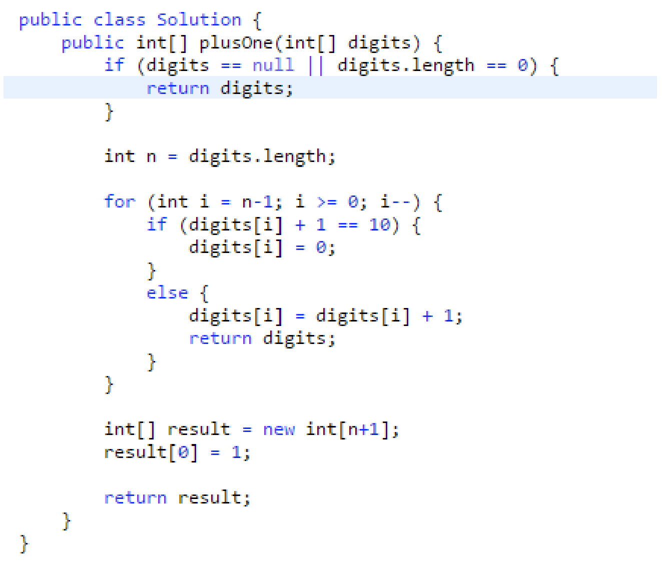
```
Input: [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
```

Analysis:

--- For each element, if elem + 1 = 10, then elem = 0, pointer moves left and continues the same calculation; if elem + 1 < 10, then elem = elem + 1, return.

--- Special case: [9, 9, 9] to [1, 0, 0, 0]. If at position 0, still not return create a new integer array, with the length of n+1.

Coding in Java:



```java
public class Solution {
    public int[] plusOne(int[] digits) {
        if (digits == null || digits.length == 0) {
            return digits;
        }

        int n = digits.length;

        for (int i = n-1; i >= 0; i--) {
            if (digits[i] + 1 == 10) {
                digits[i] = 0;
            }
            else {
                digits[i] = digits[i] + 1;
                return digits;
            }
        }

        int[] result = new int[n+1];
        result[0] = 1;

        return result;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(1) or O(N).

26. Remove Duplicates from Sorted Array

Given a sorted array *nums*, remove the duplicates in-place such that each element appear only once and return the new length.

Do not allocate extra space for another array.

Analysis:
--- Typical array rearrangement problem; two pointers needed, one mover and one anchor; when nums[mover] != nums[anchor], then nums[anchor] = nums[mover], then both pointers move right, otherwise only mover moves right.

Coding in Java:

```java
public class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int anchor = 0, mover = 0;

        while (mover < nums.length) {
            if (nums[mover] != nums[anchor]) {
                nums[anchor + 1] = nums[mover];
                anchor++;
            }

            mover++;
        }

        return anchor + 1;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(1).

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:

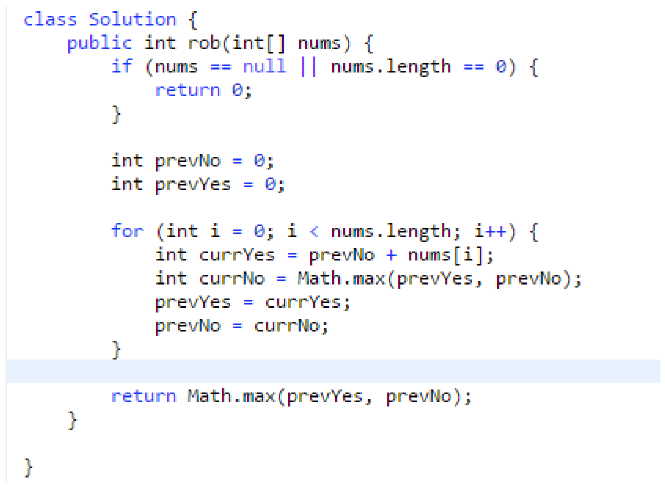
```
Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob
house 5 (money = 1).
             Total amount you can rob = 2 + 9 + 1 = 12.
```

Analysis:

--- At each house, we have a same question: Should we rob it? If yes, that means we didn't rob the previous house; if no, that means the previous house was either robbed or not robbed.

--- How much money we'll gain from a certain house? If we rob it, money = money_not_robbed_previous_house; if we not rob it, money = Math.max(money_not_robbed_previous_house, money_robbed_previous_house).

Coding in Java:



```java
class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int prevNo = 0;
        int prevYes = 0;

        for (int i = 0; i < nums.length; i++) {
            int currYes = prevNo + nums[i];
            int currNo = Math.max(prevYes, prevNo);
            prevYes = currYes;
            prevNo = currNo;
        }

        return Math.max(prevYes, prevNo);
    }

}
```

Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(1).

155. Min Stack

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

Example:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> Returns -3.
minStack.pop();
minStack.top();      --> Returns 0.
minStack.getMin();   --> Returns -2.
```

Analysis:

--- Before push a new element, compare it with the old min, if new > min, push new; if new <= min, update min, and push min, then push new. In this way, the larger value is in the inner part of the stack, and the smaller value is on the outer part.

Coding in Java:

```java
class MinStack {

    /** initialize your data structure here. */
    Stack<Integer> stack = new Stack<>();
    int min = Integer.MAX_VALUE;

    public void push(int x) {
        if (x <= min) {
            stack.push(min);
            min = x;
        }

        stack.push(x);
    }

    public void pop() {
        if (stack.pop() == min) {
            min = stack.pop();
        }
    }

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

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

Complexity analysis:

--- Time complexity:

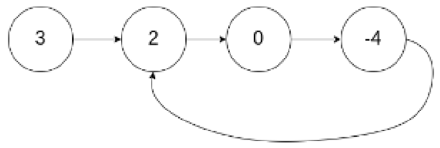
--- Space complexity:

141. Linked List Cycle

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

Example 1:

```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where tail connects
to the second node.
```



Analysis:

--- Typical hash set problem.

Coding in Java:

```java
public class Solution {
    public boolean hasCycle(ListNode head) {
        Set<ListNode> set = new HashSet<>();

        while (head != null) {
            if (set.contains(head)) {
                return true;
            }
            else {
                set.add(head);
            }

            head = head.next;
        }

        return false;
    }
}
```

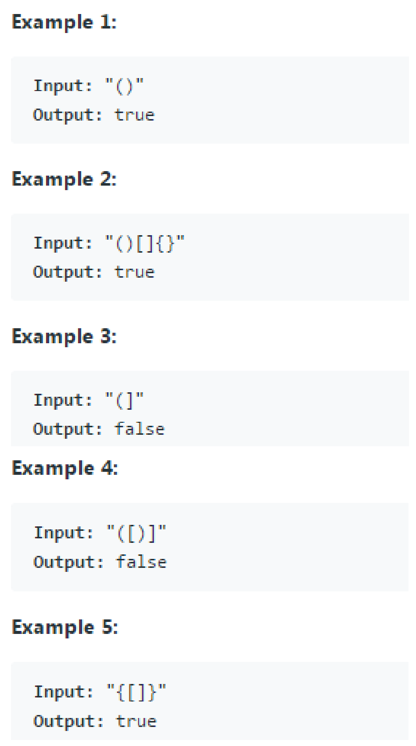
Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(N).

20. Valid Parentheses

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



Example 1:

```
Input: "()"
Output: true
```

Example 2:

```
Input: "()[]{}"
Output: true
```

Example 3:

```
Input: "(]"
Output: false
```

Example 4:

```
Input: "([)]"
Output: false
```

Example 5:

```
Input: "{[]}"
Output: true
```

Analysis:

--- At each position, if “(”, then stack pushed “)”, if “[”, then “]”, if “{”, then “}”. If “)” or “]” or “}”, then check if same with the popped. If same, then continue; if not same or stack empty, return false. At the end, if stack is not empty, like “()()]”, return false.

Coding in Java:

```
public class Solution {
    public boolean isValid(String s) {
        if (s == null || s.length() == 0) {
            return true;
        }

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

        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                stack.push(')');
            }
            else if (s.charAt(i) == '[') {
                stack.push(']');
            }
            else if (s.charAt(i) == '{') {
                stack.push('}');
            }
            else if (stack.isEmpty() || (!stack.isEmpty() && s.charAt(i) != stack.pop())) {
                return false;
            }
        }

        return stack.isEmpty();
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(N).

234. Palindrome Linked List

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

Example 1:

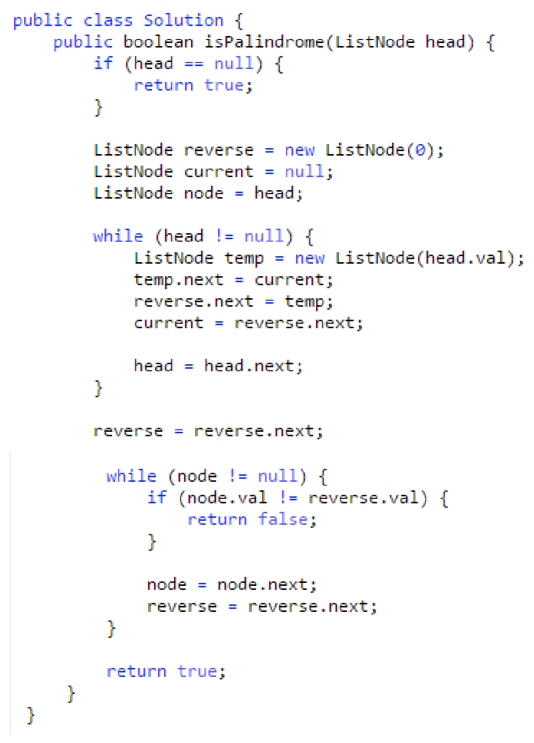
```
Input: 1->2
Output: false
```

Example 2:

```
Input: 1->2->2->1
Output: true
```

Analysis:
--- Pretty obvious, reverse the linked list.

Coding in Java:



```java
public class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null) {
            return true;
        }

        ListNode reverse = new ListNode(0);
        ListNode current = null;
        ListNode node = head;

        while (head != null) {
            ListNode temp = new ListNode(head.val);
            temp.next = current;
            reverse.next = temp;
            current = reverse.next;

            head = head.next;
        }

        reverse = reverse.next;

         while (node != null) {
             if (node.val != reverse.val) {
                 return false;
             }

             node = node.next;
             reverse = reverse.next;
         }

         return true;
     }
 }
```

Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(1) --- is reversing a linked list considered O(1)???

88. Merge Sorted Array

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

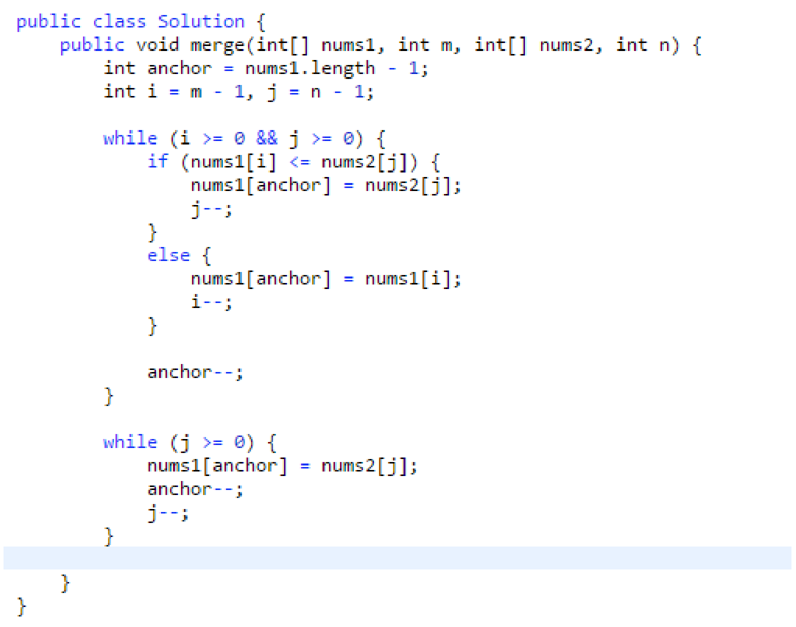
Note:

- The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.
- You may assume that *nums1* has enough space to hold additional elements from *nums2*.

Analysis:

--- Typical array rearrangement problem; two pointers, fast and slow, and one anchor pointer needed; However, worth noting that if we start from the head, some elements in the middle might be overwritten. So, we should start from the tail, and move backwards.

Coding in Java:



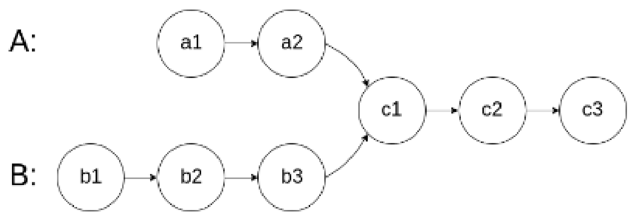
```java
public class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int anchor = nums1.length - 1;
        int i = m - 1, j = n - 1;

        while (i >= 0 && j >= 0) {
            if (nums1[i] <= nums2[j]) {
                nums1[anchor] = nums2[j];
                j--;
            }
            else {
                nums1[anchor] = nums1[i];
                i--;
            }

            anchor--;
        }

        while (j >= 0) {
            nums1[anchor] = nums2[j];
            anchor--;
            j--;
        }

    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(1).

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.

Analysis:

--- Typical two pointers problem.

Coding in Java:

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = 0, lenB = 0;

        ListNode nodeA = headA;
        ListNode nodeB = headB;

        while (headA != null) {
            lenA++;
            headA = headA.next;
        }

        while (headB != null) {
            lenB++;
            headB = headB.next;
        }

        while (lenA < lenB) {
            nodeB = nodeB.next;
            lenB--;
        }

        while (lenA > lenB) {
            nodeA = nodeA.next;
            lenA--;
        }

        while (lenA > lenB) {
            nodeA = nodeA.next;
            lenA--;
        }

        while (nodeA != null && nodeB != null) {
            if (nodeA == nodeB) {
                return nodeA;
            }

            nodeA = nodeA.next;
            nodeB = nodeB.next;
        }


        return null;
    }
}
```

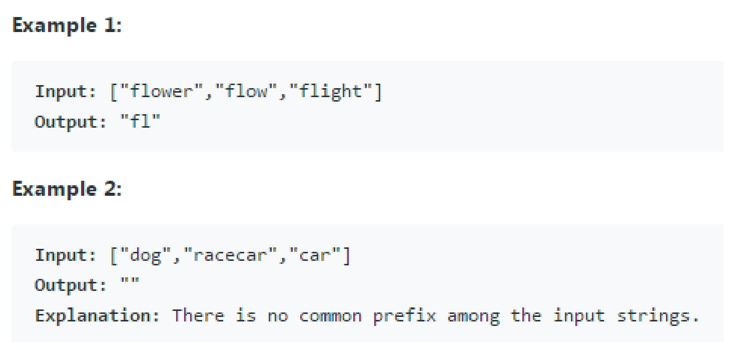
Complexity analysis:

--- Time complexity: O(m + n)
--- Space complexity: O(1).

14. Longest Common Prefix

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

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



Example 1:

```
Input: ["flower","flow","flight"]
Output: "fl"
```

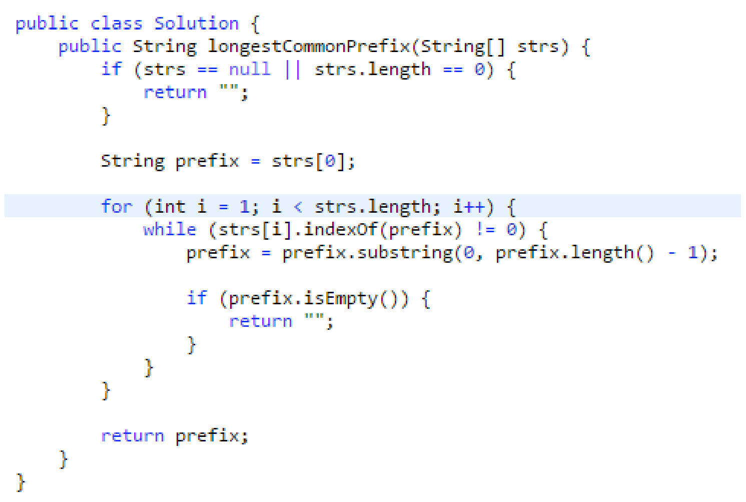
Example 2:

```
Input: ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.
```

Analysis:

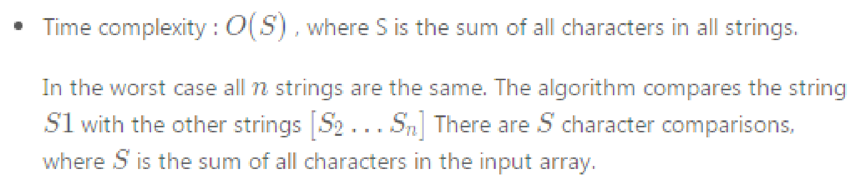
--- Java method *indexOf(String str)* returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.

Coding in Java:



```java
public class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }

        String prefix = strs[0];

        for (int i = 1; i < strs.length; i++) {
            while (strs[i].indexOf(prefix) != 0) {
                prefix = prefix.substring(0, prefix.length() - 1);

                if (prefix.isEmpty()) {
                    return "";
                }
            }
        }

        return prefix;
    }
}
```

Complexity analysis:
--- Time complexity:



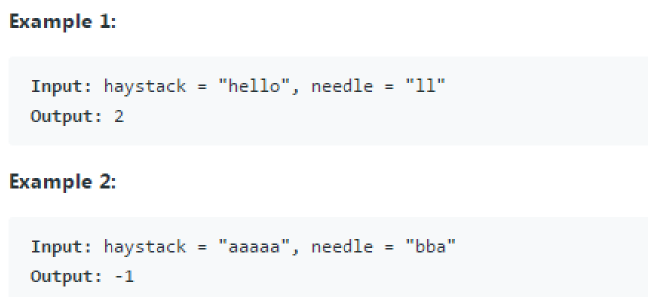
- Time complexity : $O(S)$, where S is the sum of all characters in all strings.

  In the worst case all n strings are the same. The algorithm compares the string $S1$ with the other strings $[S_2 \ldots S_n]$ There are S character comparisons, where S is the sum of all characters in the input array.

--- Space complexity: O(1).

28. Implement strStr()

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.



Example 1:

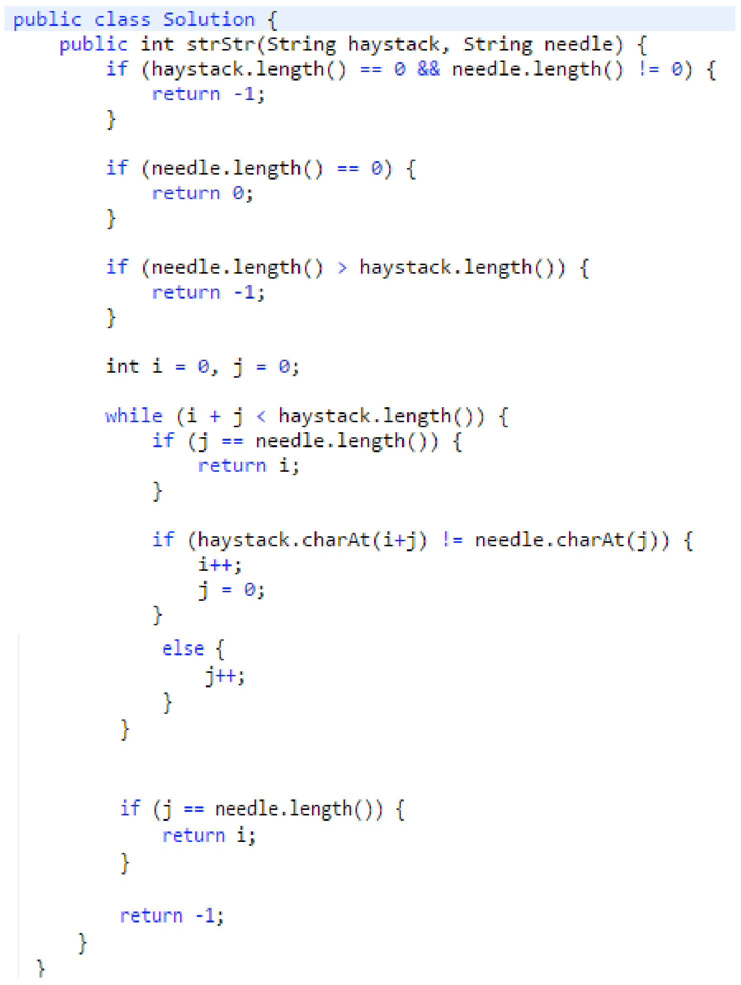
```
Input: haystack = "hello", needle = "ll"
Output: 2
```

Example 2:

```
Input: haystack = "aaaaa", needle = "bba"
Output: -1
```

Analysis:
--- Two pointers problem, but slightly different. While pointer1 is *i*, pointer2 is *i*+*j*, while *j* is pointer2_position – pointer1_position.

Coding in Java:



```java
public class Solution {
    public int strStr(String haystack, String needle) {
        if (haystack.length() == 0 && needle.length() != 0) {
            return -1;
        }

        if (needle.length() == 0) {
            return 0;
        }

        if (needle.length() > haystack.length()) {
            return -1;
        }

        int i = 0, j = 0;

        while (i + j < haystack.length()) {
            if (j == needle.length()) {
                return i;
            }

            if (haystack.charAt(i+j) != needle.charAt(j)) {
                i++;
                j = 0;
            }
             else {
                 j++;
             }
         }


         if (j == needle.length()) {
             return i;
         }

         return -1;
     }
 }
```

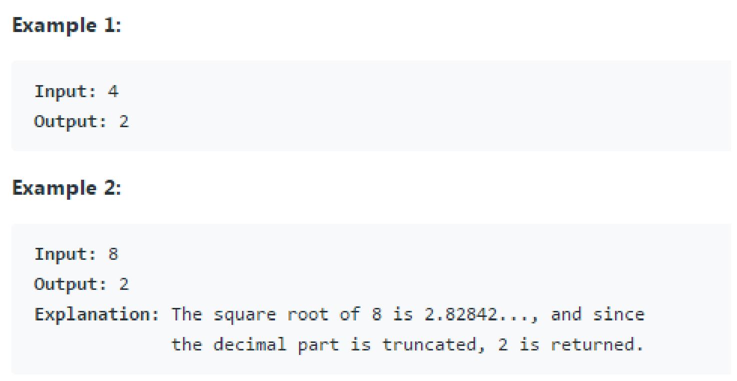
Complexity analysis:

--- Time complexity: O(N)?

--- Space complexity: O(1).

60. Sqrt(x)

Implement *int sqrt(int x).*


Example 1:

```
Input: 4
Output: 2
```

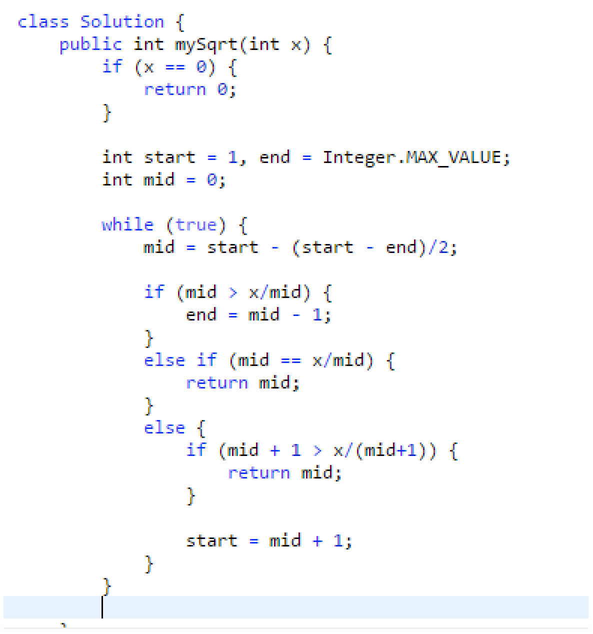
Example 2:

```
Input: 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since
             the decimal part is truncated, 2 is returned.
```

Analysis:

--- Binary search.

Coding in Java:



```java
class Solution {
    public int mySqrt(int x) {
        if (x == 0) {
            return 0;
        }

        int start = 1, end = Integer.MAX_VALUE;
        int mid = 0;

        while (true) {
            mid = start - (start - end)/2;

            if (mid > x/mid) {
                end = mid - 1;
            }
            else if (mid == x/mid) {
                return mid;
            }
            else {
                if (mid + 1 > x/(mid+1)) {
                    return mid;
                }

                start = mid + 1;
            }
        }
```

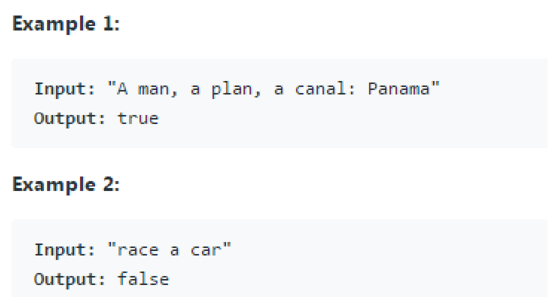
Complexity analysis:

--- Time complexity: ???

--- Space complexity: O(1).

125. Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.



Example 1:

```
Input: "A man, a plan, a canal: Panama"
Output: true
```

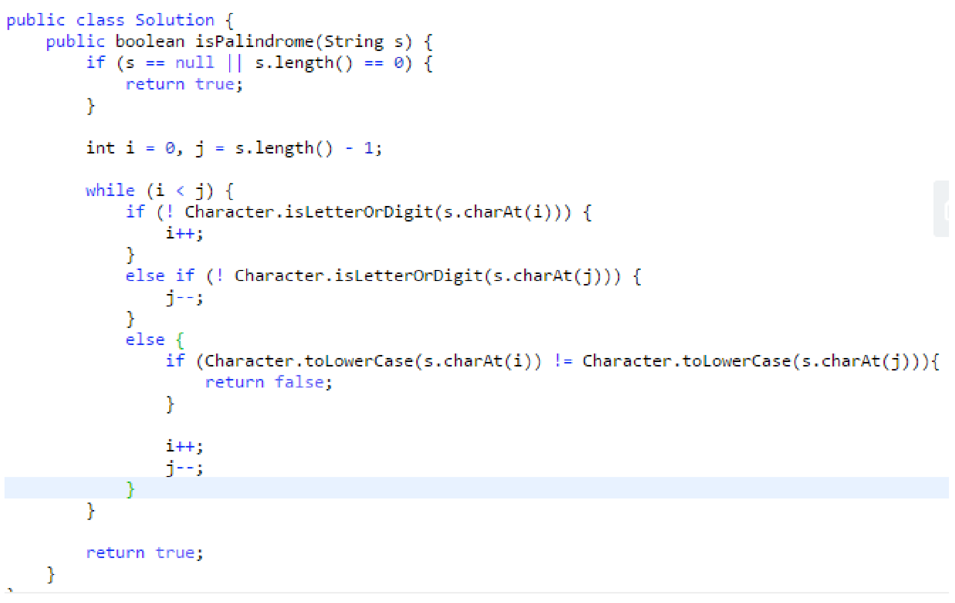
Example 2:

```
Input: "race a car"
Output: false
```

Analysis:

--- Two pointers problem, head and tail.

Coding in Java:



```java
public class Solution {
    public boolean isPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return true;
        }

        int i = 0, j = s.length() - 1;

        while (i < j) {
            if (! Character.isLetterOrDigit(s.charAt(i))) {
                i++;
            }
            else if (! Character.isLetterOrDigit(s.charAt(j))) {
                j--;
            }
            else {
                if (Character.toLowerCase(s.charAt(i)) != Character.toLowerCase(s.charAt(j))){
                    return false;
                }

                i++;
                j--;
            }
        }

        return true;
    }
```

Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(1).

189. Rotate Array

Given an array, rotate the array to the right by *k* steps, where *k* is non-negative.

Example 1:

```
Input: [1,2,3,4,5,6,7] and k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Example 2:

```
Input: [-1,-100,3,99] and k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
```

Analysis:

--- Switch problem; a temp variable needed to store the element to be switched.

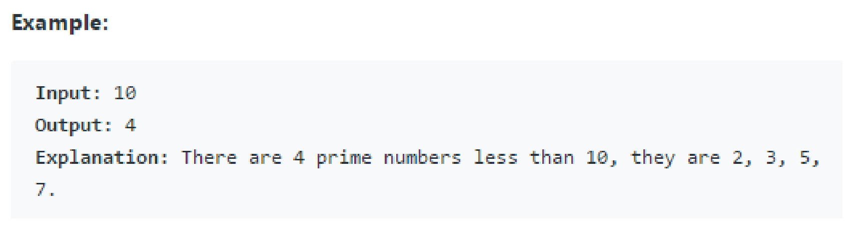
Coding in Java:

```java
class Solution {
    public void rotate(int[] nums, int k) {
        if (nums == null || nums.length == 0) {
            return;
        }

        int n = nums.length;

        while (k > 0) {
            int temp = nums[n-1];

            for (int i = n-1; i >= 1; i--) {
                nums[i] = nums[i-1];
            }

            nums[0] = temp;

            k--;
        }

    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(1).

204. Count Primes

Count the number of prime numbers less than a non-negative number, n.



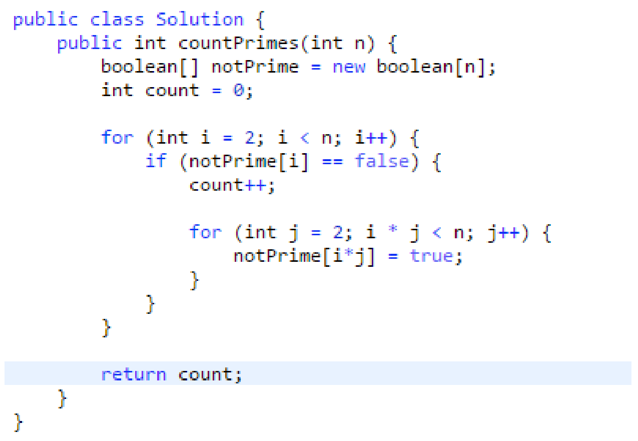
Example:

```
Input: 10
Output: 4
Explanation: There are 4 prime numbers less than 10, they are 2, 3, 5,
7.
```

Analysis:
--- For example, n = 10.
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[Prime, Prime, Prime, Prime, Prime, Prime, Prime, Prime, Prime, Prime]

@1 --- Skip;
@2 --- Prime. 2 * 2 = 4 - Not Prime, 2 * 3 = 6 - Not Prime, 2 * 4 = 8 - Not Prime, 2 * 5 = 10 - Not Prime.
--- [Prime, Prime, Prime, Not Prime, Prime, Not Prime, Prime, Not Prime, Prime, Not Prime]
@3 --- Prime. 3 * 3 = 9 – Not Prime.
--- [Prime, Prime, Prime, Not Prime, Prime, Not Prime, Prime, Not Prime, Not Prime, Not Prime]
@4 --- Not Prime;
@5 --- Prime.
@6 --- Not Prime.
@7 --- Prime.
@8 --- Not Prime.
@9 --- Not Prime.
@10 --- Not Prime.

Coding in Java:

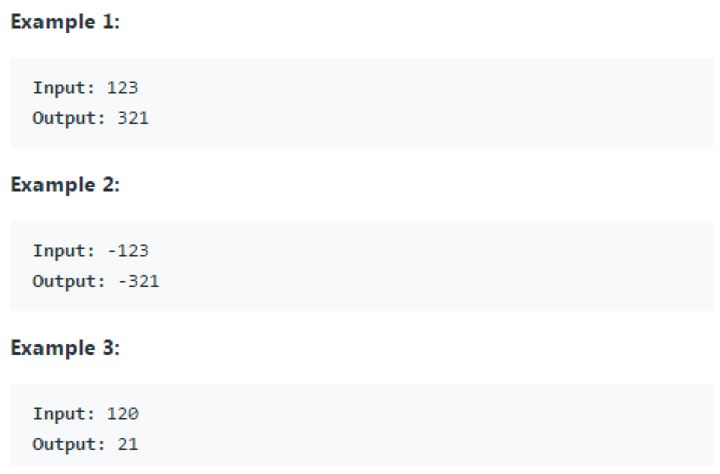


```java
public class Solution {
    public int countPrimes(int n) {
        boolean[] notPrime = new boolean[n];
        int count = 0;

        for (int i = 2; i < n; i++) {
            if (notPrime[i] == false) {
                count++;

                for (int j = 2; i * j < n; j++) {
                    notPrime[i*j] = true;
                }
            }
        }

        return count;
    }
}
```

Complexity analysis:
--- Time complexity: O(N)
--- Space complexity: O(N).

7. Reverse Integer

Given a 32-bit signed integer, reverse digits of an integer.



Example 1:

```
Input: 123
Output: 321
```

Example 2:

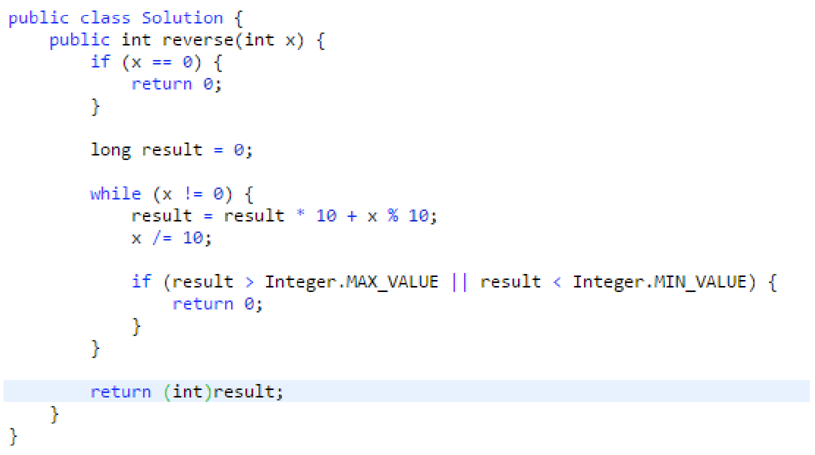
```
Input: -123
Output: -321
```

Example 3:

```
Input: 120
Output: 21
```

Analysis:
--- Pretty obvious. One thing worth noting: in case of overflow, we should add *if x > Integer.MAX_VALUE or x < Integer.MIN_VALUE….*

Coding in Java:



```java
public class Solution {
    public int reverse(int x) {
        if (x == 0) {
            return 0;
        }

        long result = 0;

        while (x != 0) {
            result = result * 10 + x % 10;
            x /= 10;

            if (result > Integer.MAX_VALUE || result < Integer.MIN_VALUE) {
                return 0;
            }
        }

        return (int)result;
    }
}
```

Complexity analysis:

--- Time complexity: O(N)

--- Space complexity: O(1).