*to-do list 2020.5.13->8.30**

top interview

179. Largest Number
130. Surrounded Regions
98. Validate Binary Search Tree
15. 3Sum
91. Decode Ways
166. Fraction to Recurring Decimal
29. Divide Two Integers
8. String to Integer (atoi)
42. Trapping Rain Water
297. Serialize and Deserialize Binary Tree
128. Longest Consecutive Sequence
295. Find Median from Data Stream

# 注意点

### backtracking

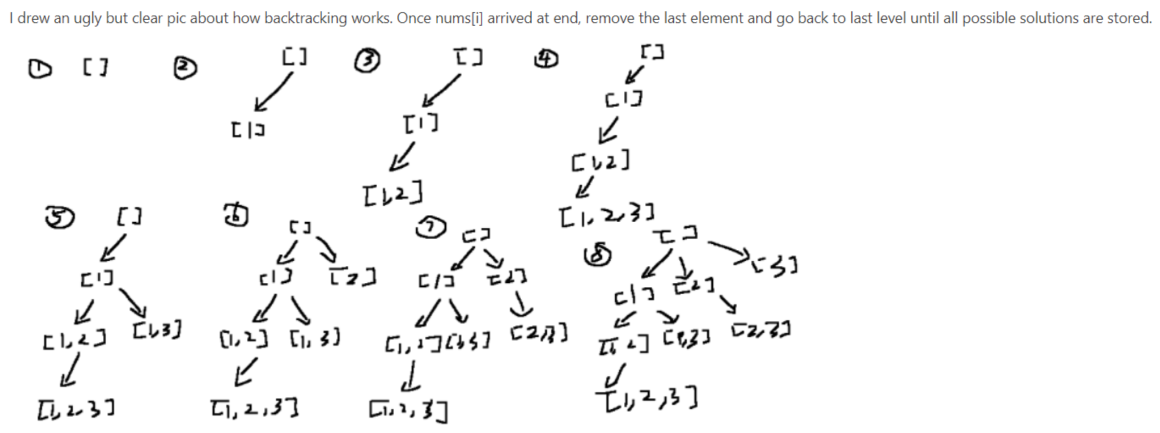
We reach our goal because of the recurse step.

1. Make a change - this reduces the input size
2. Recurse on the subproblem - This keeps reducing input size until either we reach our goal or until the input is invalid
3. Undo the change - This lets us explore other possibilties.

Visualize it as exploring a binary tree.

1. You start at the root
2. You explore the left subtree (this is equivalent to the recurse step)
3. You undo the change (once you have finished exploring the left subtree you are back to the root - now you are ready to explore the right subtree)

### Subset回溯法问题图解


I drew an ugly but clear pic about how backtracking works. Once nums[i] arrived at end, remove the last element and go back to last level until all possible solutions are stored.

# Else

IP -> int

```
    public static int IPtoInt(String IP) {
        String[] slices = IP.split("\\.");
        int res = 0;
        for (int i = 0; i < slices.length; i++) {
            res <<= 8;
            res |= Integer.parseInt(slices[i]);
        }
        return res;
    }
```

int -> IP

```
   public static String IntToIP(int n) {
        StringBuilder builder = new StringBuilder();
        for (int i = 24; i >= 0 ; i -= 8) {
            builder.append(String.valueOf((n >> i) & 0x000000FF)).append(".");
            System.out.println(n);
        }
        builder.delete(builder.length() - 1, builder.length());
        return builder.toString();
    }
```

## 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"]
```

```
class Solution {
    public void reverseString(char[] s) {
        // two pointers交换对应元素
        // 如果是String类可以先 .toCharArray()
        for (int i = 0, j = s.length - 1; i < j; i++, j--) {
            char temp = s[i];
            s[i] = s[j];
            s[j] = temp;
        }
    }
}
```

# 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]`,

image-20200516192935014

return its depth = 3.

```
public Solution {
    public int maxDepth(TreeNode root) {
        // 递归边界：若当前结点为空引用则树的最大深度为0
        if (root == null) {
            return 0;
        }
        // 递归式：否则返回当前树的最大深度 = 1 + 左右字数中深度的较大值
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
}
```

# 136. Single Number

Given a **non-empty** array of integers, every element appears *twice* except for one. Find that single one.

**Note:**

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

**Example 1:**

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

```
class Solution {
    public int singleNumbers(int[] nums) {
        int ans = 0;
        // 异或运算：两个相同的数的异或运算的结果等于0
        // 0与任何数异或运算的结果又等于该数本身
        for (int num : nums) {
            ans ^= num;
        }
        return ans;
    }
}
```

# 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"
]
```

```java
class Solution {
    public List<String> fizzBuzz(int n) {
        List<String> ret = new ArrayList<>(n);
        for (int i = 1, fizz = 0, buzz = 0; i <= n; i++) {
            fizz++;
            buzz++;
            if (fizz == 3 && buzz == 5) {
                // 同时是3和5的倍数
                ret.add("FizzBuzz");
                fizz = 0;
                buzz = 0;
            } else if (fizz == 3) {
                // 是3的倍数
                ret.add("Fizz");
                fizz = 0;
            } else if (buzz == 5) {
                // 是5的倍数
                ret.add("Buzz");
                buzz = 0;
            } else {
                // 其他的数
                ret.add(Integer.toString(i));
            }
        }
        return ret;
    }
}
```

## 206. Reverse Linked List

Reverse a singly linked list.

**Example:**

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

```
class Solution {
    // iteratively
    public ListNode reverseList(ListNode head) {
        ListNode newHead = null;
        ListNode next = null;
        whlie (head != null) {
            // 记录下一个要反转的结点
            next = head.next;
            // 改变当前结点的后继结点
            head.next = newHead;
            // 继续向后遍历
            newHead = head;
            head = next;
        }
        // 返回产生的新链表的表头
        return newHead;
    }

    // recursively
    public ListNode reverseList(ListNode head) {
        return helper(head, null);
    }

    private ListNode helper(ListNode head, ListNode newHead) {
        // 递归边界
        if (head == null) {
            return newHead;
        }
        ListNode next = head.next;
        head.next = newHead;
        // 递归式：缩小问题规模
        return helper(next, head);
        // 实际上通过传参完成了head = next;newHead = head;
    }
}
```

# 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.

**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.
```

```
class Solution {
    public void deleteNode(ListNode node) {
        // 将后继结点的值域赋值给当前结点的值域
        node.val = node.next.val;
        // 把后继结点从链表中删去
        node.next = node.next.next;
    }
}
```

# 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 majorElement(int[] nums) {
        // 先设第一个元素为主元素
        int major = nums[0];
        int count = 1;
        for (int i = 0; i < nums.length; i++) {
            // 若count为0, 则重新设当前元素为主元素
            if (count == 0) {
                major = nums[i];
                count = 1;
            // 若当前访问到的元素与主元素相等,count++
            } else if (nums[i] == major) {
                count++;
            // 若当前访问到的元素是其他元素, count--
            } else {
                count--;
            }
        }
        return major;
    }
}
```

## 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.

```
class Solution {
    public void moveZeroes(int[] nums) {
        // 边界情况
        if (nums == null || nums.length == 0) {
            return;
        }
        // 插入点
        int insertPos = 0;
        // 遍历表中元素
        for (int i = 0; i < nums.length; i++) {
            // 若当前元素不为0, 则将其插入新形成的序列尾
            if (nums[i] != 0) {
                nums[insertPos++] = nums[i];
            }
        }
        // 将剩余元素置为0
        while (insertPos < nums.length) {
            nums[insertPos++] = 0;
        }
    }
}
```

# 108. Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

**Example:**

```
Given the sorted array: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following
height balanced BST:

      0
     / \
   -3   9
   /   /
 -10  5
 /**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
```

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }
    // 辅助方法
    private TreeNode helper(int[] nums, int low, int high) {
        // 递归边界
        if (low > high) {
            return null;
        }
        // 每次都选择中间的结点作为二叉树的根结点
        // 使得左右子树的高度差异<=1
        int mid = low + (high - low) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        // 递归式
        // 递归构造根结点的左子树和右子树
        root.left = helper(nums, low, mid - 1);
        root.right = helper(nums, mid + 1, high);
        // 返回根结点
        return root;
    }
}
```

## 242. Valid Anagram

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

(An **anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.[1] For example, the word *anagram* can be rearranged into *nag a ram*, or the word *binary* into *brainy* or the word *adobe* into *abode*.)

**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.

**Follow up:**
What if the inputs contain unicode characters? How would you adapt your solution to such case?

```
class Solution {
    public boolean isAnagram(String s, String t) {
        // 若长度不同则不可能是相同字母异序词
        if (s.length() != t.length()) {
            return false;
        }
        // 用一个整形数组记录各自小写字母的出现次数
        int[] count = new int[26];
        for (int i = 0; i < s.legnth(); i++) {
            // 用两种不同的状态来表示某字母是否在字符串中出现
            // 若该字母在字符串s中出现，则将该字母所在的位置数值+1
            count[s.charAt(i) - 'a']++;
            // 若该字母在字符串t中出现，则将该字母所在的位置数值-1
            count[t.charAt(i) - 'a']--;
        }
        for (int i = 0; i < count.length; i++) {
            // 若存在字母的数值不为0，说明两字符串中字母的出现次数不一
            if (count[i] != 0) {
                return false;
            }
        }
        return true;
    }
}
```

# 122. Best Time to Buy and Sell Stock II

Say you have an array `prices` 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 (i.e., buy one and sell one share of the stock multiple times).

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

**Example 1:**

```
Input: [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1
= 4.
             Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit
= 6-3 = 3.
```

**Example 2:**

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1
= 4.
             Note that you cannot buy on day 1, buy on day 2 and sell them later,
as you are
             engaging multiple transactions at the same time. You must sell
before buying again.
```

```
class Solution {
    public int maxProfit(int[] prices) {
        // 找到每一个最长的上升序列，所得值为序列尾-序列头
        // 而这个值又与每个上升间隙的差值累积相同
        int sum = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1]) {
                sum += prices[i] - prices[i - 1];
            }
        }
        return sum;
    }
}
```

So the best solution is finding the max difference of prices in a long enough time, otherwise we will have some lose due to the rule. For example, `[2, 4, 6, 9]`(the max profit should be `9 - 2 = 7`), if we make transaction every 2 days, then profit is `(4 - 2) + (9 - 6) = 5` and the lose comes from the profit between day 3 and day 2 due to the rule, i.e. `6 - 4 = 2`

# 217. Contains Duplicate

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

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
```

```
class Solution {
    // time complexity: O(n), memory:O(n)
    public boolean containsDuplicate(int[] nums) {
        // 用一个散列表
        Set<Integer> hash = new HashSet<>();
        for (int num : nums) {
            // 如果当前元素在散列表中已经存在
            if (hash.contains(num)) {
                return true;
            }
            // 若当前元素不在散列表中则将其加入
            hash.add(num);
        }
        return false;
    }
    // time complexity: O(nlogn), memory:O(1)
    public boolean containsDuplicate(int[] nums) {
        Arrays.sort(nums);
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] == nums[i - 1]) {
                return true;
            }
        }
        return -1;
    }
}
```

# 13. Roman to Integer

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

```
Symbol       Value
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

For example, two is written as `II` in Roman numeral, just two one's added together. Twelve is written as, `XII`, which is simply `X` + `II`. The number twenty seven is written as `XXVII`, which is `XX` + `V` + `II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer. Input is guaranteed to be within the range from 1 to 3999.

**Example 1:**

```
Input: "III"
Output: 3
```

**Example 2:**

```
Input: "IV"
Output: 4
```

**Example 3:**

```
Input: "IX"
Output: 9
```

```java
class Solution {
    public int romanToInt(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }
        // 建立一个从字符到整数的映射结构
        Map<Character, Integer> map = new HashMap<>();
        map.put('I', 1);
        map.put('V', 5);
        map.put('X', 10);
        map.put('L', 50);
        map.put('C', 100);
        map.put('D', 500);
        map.put('M', 1000);
        int sum = 0;
        // 前面一个字符的数值
        int prev = map.get(s.charAt(0));
        for (int i = 1; i < s.length(); i++) {
            // 当前字符的数值
            int next = map.get(s.charAt(i));
            // 比较前后字符大小关系来确定加减
            if (prev < next) {
                sum -= prev;
            } else {
                sum += prev;
            }
            // prev向后移动
            prev = next;
        }
        // 最后一个字符的值直接加入
        sum += prev;
        return sum;
    }
}
```

# 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
    ...
```

**Example 1:**

```
Input: "A"
Output: 1
```

**Example 2:**

```
Input: "AB"
Output: 28
```

```java
class Solution {
    public int titleToNumber(String s) {
        int sum = 0;
        // 相应位置的值乘以自身的权重
        for (int i = 0; i < s.length(); i++) {
            sum = sum * 26 + (s.charAt(i) - 'A' + 1);
        }
        return sum;
    }
}
```

## 387. First Unique Character in a String

Given a string, find the first non-repeating character in it and return it's 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.

```java
class Solution {
    public int firstUniqChar(String s) {
        // 用整型数组记录每个字符出现的次数
        int[] freq = new int[256];
        for (int i = 0; i < s.length(); i++) {
            freq[s.charAt(i)]++;
        }
        // 再从头到尾遍历字符串
        for (int i = 0; i < s.length(); i++) {
            // 若当前字符串的出现次数为1
            // 则该字符时字符串中第一个不重复的字符，返回相应的下标
            if (freq[s.charAt(i)] == 1) {
                return i;
            }
        }
        return -1;
    }
}
```

# 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
```

```java
class Solution {
    // iteratively
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 建一个空结点方便操作
        ListNode head = new ListNode(0);
        // 记录下空结点的引用
        ListNode retHead = head;
        // 当两个链表都为遍历完时
        while (l1 != null && l2 != null) {
            // 挑选值较小的结点并向后遍历
            if (l1.val < l2.val) {
                head.next = l1;
                l1 = l1.next;
            } else {
                head.next = l2;
                l2 = l2.next;
            }
            head = head.next;
        }
        // 若剩下一个链表未遍历完，直接将其加入链表中
        if (l1 == null) head.next = l2;
        if (l2 == null) head.next = l1;
        // 返回新形成的链表的头结点
        return retHead.next;
    }
    // recursively
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 递归边界
        if (l1 == null) return l2;
        if (l2 == null) return l1;

        // 递归式
        if (l1.val < l2.val) {
            // 递归构造后面的结点
            l1.next = mergeTwoLists(l1.next, l2);
            // 挑选较小的结点作为返回结点
            return l1;
        } else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```

## 118. Pascal's Triangle

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

In Pascal's triangle, each number is the sum of the two numbers directly above it.

**Example:**

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

```
class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> allRows = new ArrayList<>();
        for (int i = 0; i < numRows; i++) {
            List<Integer> row = new ArrayList<>();
            // 构造每行中的元素
            for (int j = 0; j < i + 1; j++) {
                if (j == 0 || j == i) {
                    // 若为行中两侧，则填上1
                    row.add(1);
                } else {
                    // 若为中间的元素，其值为上一行中与自己相邻的两个元素的和
                    int a  = allRows.get(i - 1).get(j - 1);
                    int b = allRow.get(i - 1).get(j);
                    row.add(a + b);
                }
            }
            // 将构造好的一行加入二维数组中
            allRows.add(row);
        }
        return allRows;
    }
}
```

## 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
```

**Note**:
Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

```
class Solution {
    public int missingNumber(int[] nums) {
        // 初值设为数组长度
        int res = nums.length;
        // 用异或运算找出那个单独的数（利用下标）
        for (int i = 0; i < nums.length; i++) {
            res = res ^ i ^ nums[i];
        }
        return res;
    }
}
```

# 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.

```
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        // 用一个动态数组存储结果
        List<Integer> list = new ArrayList<>();
        // 映射结构存储第一个数组中元素到元素出现次数的映射
        Map<Integer, Integer> map = new HashMap<>();
        // 遍历第一个数组，记录相应元素出现次数
        for (int num : nums1) {
            if (map.containsKey(num)) {
                map.put(num, map.get(num) + 1);
            } else {
                map.put(num, 1);
            }
        }
        // 遍历第二个数组，求交集
        for (int num : nums2) {
            if (map.containsKey(num) && map.get(num) > 0) {
                list.add(num);
                map.put(num, map.get(num) - 1);
            }
        }
        int[] res = new int[list.size()];
        for (int i = 0; i < list.size(); i++) {
            res[i] = list.get(i);
        }
        return res;
    }
}
```

# 371. Sum of Two Integers

Calculate the sum of two integers *a* and *b*, but you are **not allowed** to use the operator `+` and `-`.

**Example 1:**

```
Input: a = 1, b = 2
Output: 3
```

**Example 2:**

```
Input: a = -2, b = 3
Output: 1
```

```
class Solution {
    public int getSum(int a, int b) {
        // 若其中有一个为0，则直接返回另一个
        if (a == 0) return b;
        if (b == 0) return a;
        // 当无需进位时退出循环
        while (b != 0) {
            // 且运算求出进位
            int carry = a & b;
            // 异或运算求出原位
            a = a ^ b;
            // 因为是进位，所以要移位
            b = carry << 1;
            // 下一轮循环计算出原位与进位的相加结果
        }
        // 返回保存的值
        return a;
    }
}
```

# 121. Best Time to Buy and Sell Stock

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

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

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

**Example 1:**

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1
= 5.
             Not 7-1 = 6, as selling price needs to be larger than buying price.
```

**Example 2:**

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

```java
class Solution {
    public int maxProfit(int[] prices) {
        // 当前最小价格，最大利润
        int minPrice = Integer.MAX_VALUE;
        // 若没有交易产生，则利润为0
        int maxProfit = 0;
        for (int price : prices) {
            // 先更新当前最小价格
            minPrice = Math.min(price, minPrice);
            // 再计算出当前情况下的最大利润
            maxProfit = Math.max(price - minPrice, maxProfit);
        }
        return maxProfit;
    }
}
```

# 202. Happy Number

Write an algorithm to determine if a number `n` is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1. Those numbers for which this process **ends in 1** are happy numbers.

Return True if `n` is a happy number, and False if not.

**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
```

```
class Solution {
    public boolean isHappy(int n) {
        // 类似于快慢指针的方法
        int slow = n, fast = n;
        do {
            // slow每次迭代一次
            slow = digitSquareSum(slow);
            // fast每次迭代两次
            fast = digitSquareSum(fast);
            fast = digitSquareSum(fast);
        } while (slow != fast);
        // 当fast == slow时退出循环，若其值为1则返回true，若为其他数则返回false
        return fast == 1;
    }
    // 计算一个数的各个位上的数的平方和
    private int digitSquareSum(int n) {
        int sum = 0;
        do {
            int temp = n % 10;
            sum += temp * temp;
            n /= 10;
        } while (n != 0);
        return sum;
    }
}
```

# 191. Number of 1 Bits

Write a function that takes an unsigned integer and return the number of '1' bits it has (also known as the Hamming weight).

**Example 1:**

```
Input: 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total
of three '1' bits.
```

**Example 2:**

```
Input: 00000000000000000000000010000000
Output: 1
Explanation: The input binary string 00000000000000000000000010000000 has a total
of one '1' bit.
```

```
class Solution {
    public int hammingWeight(int n) {
        int count = 0;
        for (int i = 0; i < 32; i++) {
            if ((n & 1) == 1) {
                count++;
            }
            n >>>= 1; // >>>高位填充0 >>高位填充符号位
        }
        return count;
    }
}
```

## 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?

**Note:** Given *n* will be a positive integer.

**Example 1:**

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

**Example 2:**

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

```
class Solution {
    public int climbStairs(int n) {
        // 动态规划求斐波那契数
        int[] res = new int[n + 1];
        res[0] = res[1] = 1;
        for (int i = 2; i <= n; i++) {
            res[i] = res[i - 1] + res[i - 2];
        }
        return res[n];
    }
}
```

## 101. Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

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
```

**Follow up:** Solve it both recursively and iteratively.

```
class Solution {
    // recursively
    public boolean isSymmetric(TreeNode root) {
        // 若该树为空树或者该树的左子树和右子树对称
        return root == null || helper(root.left, root.right);
    }
    private boolean helper(TreeNode left, TreeNode right) {
        // 若对称的结点中有一个为空，则另外一个也应该为空
        if (left == null || right == null) {
             return left == right;
        }
        // 若两者都不为空，则其值应当相等
        if (left.val != right.val) {
            return false;
        }
        // 比较下一层的对称结点
        return helper(left.left, right.right) && helper(left.right,
        right.left);
    }

    // iteratively
    public boolean isSymmetric(TreeNode root) {
        // 空树即为对称树，同时也避免操作空引用null.left/null.right
        if (root == null) {
             return true;
        }
        // 用栈来存储二叉树的结点
        Stack<TreeNode> stack = new Stack<>();
        // 将根结点的左右孩子结点压入栈
        stack.push(root.left);
        stack.push(root.right);
        // 循环直到栈中无元素
        while (!stack.empty()) {
            // 一次处理两个对称结点
            // 弹出并返回左右孩子结点
            TreeNode right = stack.pop();
            TreeNode left = stack.pop();
            // 若两个对称结点均为空结点则进入下一轮循环
            if (left == null && right == null) {
                 continue;
            }
            // 若其中有一个结点为空而另一个对称结点非空，或是对称结点值不同，返回false
            if (left == null || right == null || left.val != right.val) {
                return false;
            }
            // 继续压入下一层的对称结点
            stack.push(left.left);
            stack.push(right.right);
            stack.push(left.right);
            stack.push(right.left);
        }
        return true;
     }
}
```

# 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.
```

**Follow up:**

If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

```
class Solution {
    public int maxSubarray(int[] nums) {
    // 特殊情况
    if (nums == null || nums.length == 0) {
         return 0;
    }
    // sum表示子数组的和
    // max表示目前遇到过的最大的数组的和
    int sum = nums[0];
    int max = nums[0];
    for (int i = 1; i < nums.length; i++) {
        // 如果之前的和是小于0，那么没必要加上它，因为反而会使当前的子数组的和变小
        // 直接令其为0，舍弃掉它，重新开始计算子数组最大和
        sum = (sum < 0 ? 0 : sum) + nums[i];
        max = Math.max(sum, max);
    }
    return max;
    }
}
```

## 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].
```

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int[] res = new int[2];
        // 建立元素到其对应下标值的映射
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            // 先求出其所需匹配元素
            int numToFind = target - nums[i];
            // 若映射结构中存在该元素则返回其下标值(value)，与当前值的下标值
            if (map.containsKey(numToFind)) {
                res[0] = map.get(numToFind);
                res[1] = i;
                break;
            }
            map.put(nums[i], i);
        }
        return res;
    }
}
```

## 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, you must do this by **modifying the input array** in-place with O(1) extra memory.

**Example 1:**

```
Given nums = [1,1,2],

Your function should return length = 2, with the first two elements of nums being
1 and 2 respectively.

It doesn't matter what you leave beyond the returned length.
```

**Example 2:**

```
Given nums = [0,0,1,1,1,2,2,3,3,4],

Your function should return length = 5, with the first five elements of nums
being modified to 0, 1, 2, 3, and 4 respectively.

It doesn't matter what values are set beyond the returned length.
```

```
class Solution {
    public int removeDuplicates(int[] nums) {
        // 特殊情况
        if (nums == null || nums.length == 0) {
            return 0;
        }
        // 重新建表的思想
        // 将元素插入新表的位置
        // 先设数组中的第一个元素为不同的元素
        int insertPos = 0;
        for (int i = 1; i < nums.length; i++) {
            // 遍历后面的元素，若当前元素与新表中的最后一个元素不同
            // 则将其插入新表的末尾
            if (nums[i] != nums[insertPos]) {
                nums[++insertPos] = nums[i];
            }
        }
        // 返回新表的长度
        return insertPos + 1;
    }
}
```

## 38. Count and Say

The count-and-say sequence is the sequence of integers with the first five terms as following:

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

`1` is read off as `"one 1"` or `11`.
`11` is read off as `"two 1s"` or `21`.
`21` is read off as `"one 2`, then `one 1"` or `1211`.

Given an integer *n* where 1 ≤ *n* ≤ 30, generate the *n*th term of the count-and-say sequence. You can do so recursively, in other words from the previous member read off the digits, counting the number of digits in groups of the same digit.

Note: Each term of the sequence of integers will be represented as a string.

**Example 1:**

```
Input: 1
Output: "1"
Explanation: This is the base case.
```

**Example 2:**

```
Input: 4
Output: "1211"
Explanation: For n = 3 the term was "21" in which we have two groups "2" and "1",
"2" can be read as "12" which means frequency = 1 and value = 2, the same way "1"
is read as "11", so the answer is the concatenation of "12" and "11" which is
"1211".
```

```
class Solution {
    public String countAndSay(int n) {
        // 递归出口
        if (n == 1) {
            return "1";
        }
        // 递归求出上一个字符串
        String prev = countAndSay(n - 1);
        // 用非线程安全的构造我们所需的字符串
        StringBuilder str = new StringBuilder();
        // 下标
        int i = 0;
        while (i < prev.length()) {
            // 统计与当前该位的数字相同的相邻的数字的个数
            int freq = 0;
            // 当前位的数字
            char currentDigit = prev.charAt(i);
            // 统计
            while (i + freq < prev.length && prev.charAt(i + freq) ==
currentDigit) {
                freq++;
            }
            // 有几个
            str.append(freq);
            // 当前数值的数
            str.append(currentDigit);
            // 向后遍历
            i += freq;
        }
        // 转化为String类
        return str.toString();
    }
}
```

## 155. Min Stack

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

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- getMin() -- Retrieve the minimum element in the stack.

**Example 1:**

```
Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

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

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

```
class MinStack {
    // 一个栈用于作普通用途
    private Stack<Integer> stack;
    // 一个栈用于存储每个阶段的最小元素
    private Stack<Integer> min;
    public MinStack() {
        stack = new Stack<>();
        min = new Stack<>();
    }
    public void push(int n) {
        stack.push(n);
        // 若存储最小值的栈为空，或者是当前元素<=当前最小元素
        // 将其压入最小栈
        if (min.empty() || n <= min.peek()) {
            min.push(n);
        }
    }
    public void pop() {
        int popped = stack.pop();
        // 若从普通栈弹出的元素为当前最小元素相同
        // 需要将其从最小栈中弹出
        if (popped == min.peek()) {
            min.pop();
        }
    }
    public int top() {
         return stack.peek();
    }
    // 最小栈的顶部存储的就是当前最小元素
    public int getMin() {
        return min.peek();
    }
}
```

## 66. Plus One

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

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contain a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

**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.
```

```
class Solution {
    public int[] plusOne(int[] digits) {
    int len = digits.length;
    // 从低位到高位遍历
    for (int i = len - 1; i >= 0; i--) {
        // 若当前位数不为9，则加1后直接返回数组
        if (digits[i] < 9) {
            digits[i]++;
            return digits;
        } else {
            // 若当前位数为9，另其为0
            digits[i] = 0;
        }
    }
    // 特殊情况：当数组中的所有数都为9的时候，不会再for循环内返回
    int[] newNumber = new int[len + 1];
    newNumber[0] = 1;
    return newNumber;
    }
}
```

## 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
```

**Example 4:**

```
Input: 45
Output: false
```

**Follow up:**
Could you do it without using any loop / recursion?

```
class Solution {
    // iteratively
    public boolean isPowerOfThree(int n) {
        // 若传入的数大于1
        if (n > 1) {
            // 若当前数能被3整除
            while (n % 3 == 0) {
                // 除以3
                n /= 3;
            }
        }
        // 当数不能被3整除时，判断其是否为1
        return n == 1;
    }

    // recursively
    public boolean isPowerOfThree(int n) {
        // 原理同迭代，n == 1为递归出口，isPowerOfThree(n / 3)为递归式
        return n > 0 && (n == 1 || (n % 3 == 0) && isPowerOfThree(n / 3));
    }

    // using log
    public boolean isPowerOfThree(int n) {
        if (n <= 0) {
            return false;
        }
        // 如果n为3的倍数
        // 3^k = 3^(klog3/3)
        return n == Math.pow(3, Math.round(Math.log(n) / Math.log(3)));
    }
}
```

## 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: nums = [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: nums = [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.
```

```
class Solution {
    public int rob(int[] nums) {
        // 处理特殊情况
        if (nums == null || nums.length == 0) {
            return 0;
        }
        // dp数组存储当前收益的最大值
        int[] dp = new int[nums.length + 1];
        dp[1] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            // 遇到一户人家有两种选择
            // 抢：当前收益 = 继承抢前面的前面一家时候的最大收益 + 此家收益
            // 不抢：当前收益 = 直接继承抢前面一家时候的最大收益
            dp[i + 1] = Math.max(dp[i], dp[i - 1] + nums[i]);
        }
        // 返回收益的最大值
        return dp[nums.length];
    }
}
```

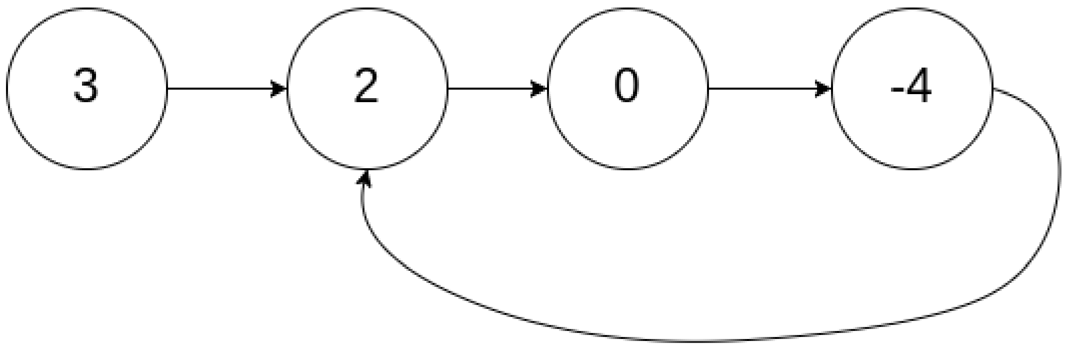
## 141. Linked List Cycle

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

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.

**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.
```



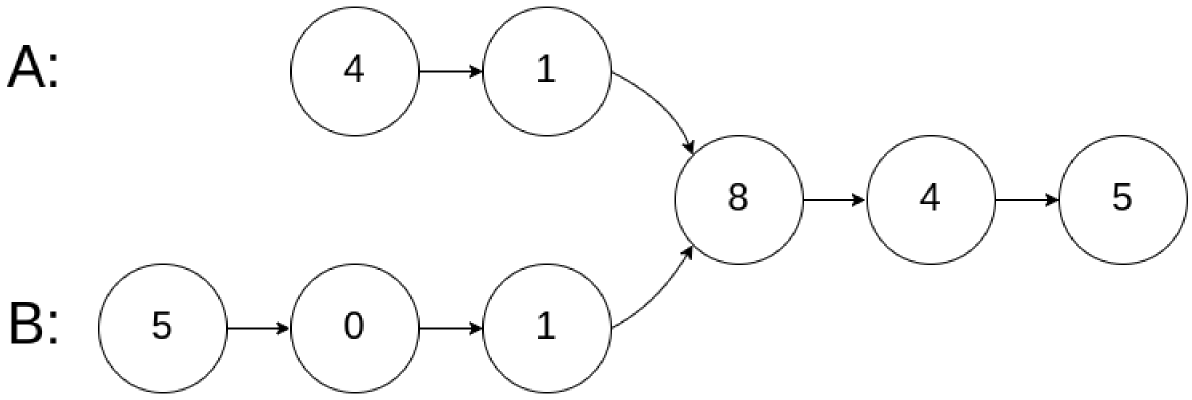
```
class Solution {
    public boolean hasCycle(ListNode head) {
        // 快慢指针的方法
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            // 类似于环形跑道，快的人终究会追上慢的人：快的人每次逼近一格
            slow = slow.next;
            fast = fast.next.next;
            // 如果两个指针能相遇说明有环
            if (slow == fast) {
                return true;
            }
        }
        // 如果快指针先到达链表尾，说明无环
        return false;
    }
}
```

## 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:

**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.
```

```
class Solution {
    // 利用调转指针（将表A的表尾和表B的表头从逻辑上串起来）的方法对齐指针
    // 设链表A长度为 a + c, 表B长度为 b + c， 调转指针后
    // 指针A走的长度为 a + c + b, 指针B走的长度为b + c + a
    // 在走了相同长的路之后，两个指针会相遇或者是同时为null（各走到另一方的表尾）
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        ListNode nodeA = headA;
        ListNode nodeB = headB;
        while (nodeA != nodeB) {
            nodeA = nodeA == null ? headB : nodeA.next;
            nodeB = nodeB == null ? headA : nodeB.next;
        }
        return nodeA;
    }

    // 利用求长度的方法来对齐指针
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        // 求长度
        int lenA = getLength(headA);
        int lenB = getLength(headB);
        // 对齐指针
        while (lenA > lenB) {
            headA = headA.next;
            lenA--;
        }
        while (lenB > lenA) {
            headB = headB.next;
            lenB--;
        }
        // 同步向后遍历，直到相遇或者同时为null
        while (headA != headB) {
            headA = headA.next;
            headB = headB.next;
        }
        return headA;
    }

    private int getLength(ListNode head) {
        int len = 0;
        while (head != null) {
            head = head.next;
            len++;
        }
        return len;
    }
}
```

# 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 (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*.

**Example:**

```
Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

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

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        // 三个索引分别指向三个表最后一个元素的位置
        int i = m - 1;
        int j = n - 1;
        int k = m + n - 1;
        // 循环直到两个表中有一个表没有元素为止
        while (i >= 0 && j >= 0) {
            // 挑选两个原表中的较大元素加入新表末尾
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        // 若第二个表中还有元素则直接让其将入
        // 若第二个表已经比较完了，那说明所有元素均已在其位置上了
        while (j >= 0) {
            nums1[k--] = nums2[j--];
        }
    }
}
```

## 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?

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        // 快慢指针
        ListNode slow = head;
        ListNode fast = head;
        // 快指针走到表尾时，慢指针走到表中间
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        // 如果表中元素个数为奇数个，将slow向后移动一格
        if (fast != null) {
            slow = slow.next;
        }
        // 逆转后半链表，并返回逆转后的头结点
        slow = reverse(slow);
        // fast指向原链表的头结点（原链表此时只有前半部分有效，后半部分已被修改）
        fast = head;
        // 遍历比较大小是否相等，以判断回文
        while (slow != null) {
            if (slow.val != fast.val) {
                return false;
            }
            // while loop 里面别忘了.next！！
            slow = slow.next;
            fast = fast.next;
        }
        return true;
    }

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

# 20. Valid Parentheses

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

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Note that an empty string is also considered valid.

**Example 1:**

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

**Example 2:**

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

**Example 3:**

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

```
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            switch(c) {
                    // 若为左括号，则压入相应的右括号
                case '[':
                    stack.push(']');
                    break;
                case '{':
                    stack.push('}');
                    break;
                case '(':
                    stack.push(')');
                    break;
                default:
                    // 若为右括号，如果栈为空，说明没有和其匹配的左括号：]()
                    // 或者弹出的元素与其不相等:(]， 均返回false
                    if (stack.empty() || stack.pop() != c) {
                        return false;
                    }
            }
        }
        // 空串也是有效的
        return stack.empty();
    }
}
```

# 172. Factorial Trailing Zeroes

Given an integer *n*, return the number of trailing zeroes in *n*!.

**Example 1:**

```
Input: 3
Output: 0
Explanation: 3! = 6, no trailing zero.
```

**Example 2:**

```
Input: 5
Output: 1
Explanation: 5! = 120, one trailing zero.
```

**Note:** Your solution should be in logarithmic time complexity.

```
class Solution {
    // iteratively
    public int trailingZeroes(int n) {
        int count = 0;
        while (n != 0) {
            count += n / 5;
            n /= 5;
        }
        return count;
    }

    // recursively
    public int trailingZeroes(int n) {
        if (n == 0) {
            return 0;
        }
        return n / 5 + trailingZeroes(n / 5);
    }
}
```

## 190. Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

**Example 1:**

```
Input: 00000010100101000001111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string 00000010100101000001111010011100 represents
the unsigned integer 43261596, so return 964176192 which its binary
representation is 00111001011110000010100101000000.
```

```
class Solution {
    public int reverseBits(int n) {
        // 存储反转bit后的n
        int res = 0;
        // 遍历32位bit
        for (int i = 0; i < 32; i++) {
            // 先取出n的最低的一位
            int end = n & 1;
            // n向右移一位（最高位填充符号位，但在这里没关系，因为不会碰到）
            n >>= 1;
            // 先左移res一位，再加上当前n的最低位
            //（相当于整型中的res*=10 要放在 res += n % 10的前面）
            res <<= 1;
            res |= end;
        }
        return res;
    }
}
```

## 125. Valid Palindrome

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

**Note:** For the purpose of this problem, we define empty string as valid palindrome.

**Example 1:**

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

**Example 2:**

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

```
class Solution {
    public boolean isPalindrome(String s) {
        for (int i = 0, j = s.length() - 1; 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;
            }
        }
        return true;
    }
}
```

## 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.
```

**Note:**

All given inputs are in lowercase letters `a-z`.

```
class Solution {
    public String longestCommonPrefix(String[] strs) {
        // 处理特殊情况
        if (strs == null || strs.length == 0) {
            return "";
        }
        // 用第一个字符串和后面的字符串进行比较
        String s = strs[0];
        for (int i = 1; i < strs.length; i++) {
            // 若后面的字符串不以第一个字符串为开头，则一直删减第一个字符串的尺寸
            // 直到满足条件或者s为空串
            while (!strs[i].startsWith(s)) {
                s = s.substring(0, s.length() - 1);
            }
        }
        return s;
    }
}
```

# 28. Implement strStr()

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
```

**Clarification:**

What should we return when `needle` is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when `needle` is an empty string. This is consistent to C's strstr() and Java's indexOf().

```
class Solution {
    public int strStr(String haystack, String needle) {
        // 若needle为空串，则返回0
        if (needle.isEmpty()) {
            return 0;
        }
        // 遍历haystack
        for (int i = 0; i < haystack.length(); i++) {
            // 若haystack已经没有足够字符进行比较，直接退出
            if (i + needle.length() > haystack.length()) {
                break;
            }
            // 比较haystack i + j位置 和 needle j位置的元素
            for (int j = 0; j < needle.length(); j++) {
                // 若不等则直接退出，从haystack的下一个字符开始继续比较
                if (haystack.charAt(i + j) != needle.charAt(j)) {
                    break;
                }
                // 若j已经到达needle最后一个字符，说明匹配完成，返回相应的i
                // 即haystack中第一个匹配的元素即可
                if (j == needle.length() - 1) {
                    return i;
                }
            }
        }
        // 若遍历完了还未匹配成功则返回-1
        return -1;
    }
}
```

## 189. Rotate Array

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

**Follow up:**

- Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
- Could you do it in-place with O(1) extra space?

**Example 1:**

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

```
class Solution {
    public void rotate(int[] nums, int k) {
        // 取余，一是为了避免数组索引越界
        // 二是因为移动k个位置和移动k + nums.length个位置所需操作一样
        k %= nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }

    // 将数组nums [start, end]范围内的元素逆置
    private void reverse(int[] nums, int start, int end) {
        for (int i = start, j = end; i < j; i++, j--) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
}
```

## 69. Sqrt(x)

Implement `int sqrt(int x)`.

Compute and return the square root of *x*, where *x* is guaranteed to be a non-negative integer.

Since the return type is an integer, the decimal digits are truncated and only the integer part of the result is returned.

**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.
```

```
class Solution {
    public int mySqrt(int x) {
        int left = 1, right = x;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (mid <= x / mid) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return left - 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.
```

```
class Solution {
    public int countPrimes(int n) {
        // 筛法
        // 利用布尔数组，初始时将0-n-1均认为是质数
        boolean[] notPrime = new boolean[n];
        // 计数器
        int count = 0;
        // 2是第一个质数，从它开始遍历
        for (int i = 2; i < n; i++) {
            // 如果当前i是质数
            if (!notPrime[i]) {
                // 计数+1
                count++;
                // 并将后面的小于n的 2*i 3*i..均标记为非质数
                for (int j = 2; i * j < n; j++) {
                    notPrime[i * j] = true;
                }
            }
        }
        return count;
    }
}
```

## 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
```

**Note:**
Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−2^31, 2^31 − 1]. For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

```
class Solution {
    public int reverse(int x) {
        // 标记符号，因为后面取余运算是针对正整数的
        int sign = x < 0 ? - 1 : 1;
        // long类型的数据存储运算结果，因为int逆置后可能会超过2^31 - 1
        long res = 0;
        x = Math.abs(x);
        while (x != 0) {
            // 先将原数*10，腾出个位
            res *= 10;
            // 取出x的最低位，并将其加入res中
            res += x % 10;
            // 继续取x的高位
            x /= 10;
        }
        // 还原符号
        res *= sign;
        // 若没有发生数据溢出则直接返回，否则返回0
        return (int)res == res ? (int)res : 0;
    }
}
```

## 46. Permutations

Given a collection of **distinct** integers, return all possible permutations.

**Example:**

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

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        // 存储全排列
        List<List<Integer>> list = new ArrayList<>();
        // 回溯法
        backtrack(list, new ArrayList<>(), nums);
        // 返回结果
        return list;
    }
    // time complexity: O(n * n!),
    // 最好用树来模拟整个过程：+（添加元素） ->（继续向下探索） -（回溯后，删减元素）
    private void backtrack(List<List<Integer>> list, List<Integer> tempList,
int[] nums) {
        // 递归出口：
        // 如果当前的tempList中的元素与数组中元素相等
        // 则表明得到了一个结果，将其中的元素加入list中，并返回到上一层中将tempList的最后一
个元素删去
        if (tempList.size() == nums.length) {
            list.add(new ArrayList<>(tempList));
            return;
        }
        // 遍历数组
        for (int i = 0; i < nums.length; i++) {
            // 若当前数组元素已存在于链表中，则往后找
            if (tempList.contains(nums[i])) continue;
            // 否则则直接添加入tempList中
            tempList.add(nums[i]);
            // 递归
            backtrack(list, tempList, nums);
            // 递归返回后，将tempList的最后一个元素删去，给探索新的元素组合留空间
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

# 94. Binary Tree Inorder Traversal

Given a binary tree, return the *inorder* traversal of its nodes' values.

**Example:**

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

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

**Follow up:** Recursive solution is trivial, could you do it iteratively?

```
class Solution {
    // recursively
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<>();
        helper(root, list);
        return list;
    }

    private void helper(TreeNode root, List<Integer> list) {
        // 若为空树，则直接返回
        if (root == null) {
            return;
        }
        // 递归遍历左子树
        helper(root.left, list);
        // 添加结点
        list.add(root.val);
        // 递归遍历右子树
        helper(root.right, list);
    }

    // iteratively
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<>();
        // 若为空树，则返回空队列
        if (root == null) {
            return list;
        }
        // 栈用于存储树结点
        Deque<TreeNode> stack = new LinkedList<>();
        // 工作指针
        TreeNode cur = root;
        while (cur != null || !stack.isEmpty()) {
            // 若当前结点不为空，则将当前结点压入栈，一直往左走
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }
            // 此时的栈顶元素即为LNR中的N
            cur = stack.pop();
            // 将其中元素加入list
            list.add(cur.val);
            // 往右走，即LNR中的R
            cur = cur.right;
        }
        return list;
    }
}
```

## 22. Generate Parentheses

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

```
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> list = new ArrayList<>();
        backtrack(list, "", 0, 0, n);
        return list;
    }
    /*
    Constraints: First, the first character should be "(". Second, at each step,
you can either print "(" or ")", but print ")" only when there are more "("s than
")"s. Stop printing out "(" when the number of "(" s hit n.
    */
    private void backtrack(List<String> list, String s, int left, int right, int
leftMax) {
        if (s.length() == 2 * leftMax) {
            list.add(s);
            return;
        }
        if (left < leftMax) {
            backtrack(list, s + "(", left + 1, right, leftMax);
        }
        if (right < left) {
            backtrack(list, s + ")", left, right + 1, leftMax);
        }
    }
}
```

## 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.
- It's guaranteed that the answer is unique, in other words the set of the top k frequent elements is unique.
- You can return the answer in any order.

```
public Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        // 遍历数组。建立数组元素到数组出现次数的映射
        for (int n : nums) {
            map.put(n, map.getOrDefault(n, 0) + 1);
        }
        // 桶：数组中存储的元素是List，在lists[i]这条链表上存储着所有出现次数为i的元素
        // 且数组的最大长度是为了保证，当nums中都是同一数字时
        // 其最大出现次数为nums.length
        List<Integer>[] lists = new List[nums.length + 1];
        // 遍历map的键值对
        for (Map.Entry<Integer, Integer> e : map.entrySet()) {
            // 数字
            Integer num = e.getKey();
            // 相应的出现次数
            Integer freq = e.getValue();
            // 若频次对应的链表还未创建则先创建
            if (lists[freq] == null) {
                lists[freq] = new ArrayList<>();
            }
            // 将数字将入到相应频次的链表中
            lists[freq].add(num);
        }
        List<Integer> res = new ArrayList<>();
        // 从后往前遍历频次数组，将数组中出现频次较高的元素加入链表
        for (int i = lists.length - 1; i >= 0 && res.size() < k; i--) {
            if (lists[i] != null) {
                res.addAll(lists[i]);
            }
        }
        int[] topk = new int[k];
        for (int i = 0; i < k; i++) {
            topK[i] = res.get(i);
        }
        return topK;
    }
}
```

## 78. Subsets

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

**Note:** The solution set must not contain duplicate subsets.

**Example:**

```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> list = new ArrayList<>();
        backtrack(list, new ArrayList<>(), nums, 0);
        return list;
    }

    private void backtrack(List<List<Integer>> list, List<Integer> tempList,
int[] nums, int start) {
        list.add(new ArrayList<>(tempList));
        // for loop
        for (int i = start; i < nums.length; i++) {
            // add
            tempList.add(nums[i]);
            // recurse
            backtrack(list, tempList, nums, i + 1);
            // remove
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

## 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]
```

**Constraint:** It's guaranteed that the product of the elements of any prefix or suffix of the array (including the whole array) fits in a 32 bit integer.

**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.)

```
class Solution {
    /*
        结果由左右两个部分构成
        24 = left: 1 * right: 2 * 3 * 4
        12 = left: 1 * right: 3 * 4
    */
    public int[] productExceptSelf(int[] nums) {
        int[] res = new int[nums.length];
        res[0] = 1;
        // 从左往右，计算左边的部分
        for (int i = 1; i < nums.length; i++) {
            res[i] = res[i - 1] * nums[i - 1];
        }
        int right = 1;
        // 从右往左，计算右边的部分
        for (int i = nums.length - 1; i >= 0; i--) {
            res[i] *= right;
            right *= nums[i];
        }
        return res;
    }
}
```

## 230. Kth Smallest Element in a BST

Given a binary search tree, write a function `kthSmallest` to find the **k**th smallest element in it.

**Example 1:**

```
Input: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
Output: 1
```

**Example 2:**

```
Input: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
Output: 3
```

**Follow up:**
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

```
class Solution {
    // 二叉搜索树中序遍历序列的第k个数，就是第k个最小的数
    // iteratively
    public int kthSmallest(TreeNode root, int k) {
        Deque<TreeNode> stack = new LinkedList<>();
        int count = 0;
        TreeNode cur = root;
        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }
            cur = stack.pop();
            if (++count == k) {
                return cur.val;
            }
            cur = cur.right;
        }
        return Integer.MIN_VALUE;
    }

    // recursively
    private int count = 0;
    private int res = Integer.MIN_VALUE;
    public int kthSmallest(TreeNode root, int k) {
        helper(root, k);
        return res;
    }

    private void helper(TreeNode root, int k) {
        if (root == null) {
            return;
        }
        helper(root.left, k);
        if (++count == k) {
            res = root.val;
        }
        // 注意这里：剪枝，已经得到了结果后就不必再想右子树遍历了
        if (count < k) {
            helper(root.right, k);
        }
    }
}
```

## 49. Group Anagrams

Given an array of strings, group anagrams together.

**Example:**

```
Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
Output:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

**Note:**

- All inputs will be in lowercase.
- The order of your output does not matter.

```
class Solution {
    // 将同字异构词组成一组
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> res = new ArrayList<>();
        if (strs == null || strs.length == 0) {
            return res;
        }
        // 建立从keyStr到一个字符串的List的映射
        // 相同的字母组成的字符串, 他们对应的keyStr是相同的
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            // 先求出字母出现频度的字符数组
            char[] count = new char[26];
            char[] c = s.toCharArray();
            for (int i = 0; i < c.length; i++) {
                count[c[i] - 'a']++;
            }
            // 再根据字符数组，创建keyStr
            String keyStr = new String(count);
            // 如果map中没有该keyStr对应的键值对，则创建一个
            if (!map.containsKey(keyStr)) {
                map.put(keyStr, new ArrayList<>());
            }
            // 将该字符串加入到对应的list中
            map.get(keyStr).add(s);
        }
        // 遍历Collection<List<String>>， 添加结果
        for (List<String> list : map.values()) {
            res.add(list);
        }
        return res;
    }
}
```

## 48. Rotate Image

You are given an *n* x *n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

**Note:**

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**

```
Given input matrix =
[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],

rotate the input matrix in-place such that it becomes:
[
  [7,4,1],
  [8,5,2],
  [9,6,3]
]
```

```java
class Solution {
    public void rotate(int[][] matrix) {
        // 求出维度
        int n = matrix.length;
        // 将矩阵沿着中间逆置，以一维数组为操作单位
        for (int i = 0, j = n - 1; i < j; i++, j--) {
            int[] temp = matrix[i];
            matrix[i] = matrix[j];
            matrix[j] = temp;
        }
        // 将矩阵沿着对角线逆置
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }
    }
}
```

# 328. Odd Even Linked List

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

**Example 1:**

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

**Example 2:**

```
Input: 2->1->3->5->6->4->7->NULL
Output: 2->3->6->7->1->5->4->NULL
```

**Constraints:**

- The relative order inside both the even and odd groups should remain as it was in the input.
- The first node is considered odd, the second node even and so on ...
- The length of the linked list is between `[0, 10^4]`.

```
class Solution {
    public ListNode oddEvenList(ListNode head) {
        // 不管head为不为null,最终都返回head
        if (head != null) {
            ListNode odd = head;
            ListNode even = head.next;
            // 先记录下evenList的头结点
            ListNode evenHead = even;
            while (even != null && even.next != null) {
                // odd和even都往后面的后面连接
                odd.next = odd.next.next;
                even.next = even.next.next;
                // 向后遍历
                odd = odd.next;
                even = even.next;
            }
            // 将oddList的尾鱼evenList的头连接上
            odd.next = evenHead;
        }
        return head;
    }
}
```

# 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.

```
class Solution {
    public int findDuplicate(int[] nums) {
        // 类似于用快慢指针找到链表中环的入口
        int slow = 0;
        int fast = 0;
        // 先找到两指针相遇点
        do {
            slow = nums[slow];
            fast = nums[nums[fast]];
        } while (slow != fast);
        slow = 0;
        // 两次指针再次相遇的地方就是环的入口，即重复的数字
        do {
            slow = nums[slow];
            fast = nums[fast];
        } while (slow != fast);
        return slow;
    }
}
```

# 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 {
    // 用最小堆
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        // 遍历数组中的元素
        for (int n : nums) {
            minHeap.add(n);
            // 如果堆中元素超过了k
            if (minHeap.size() > k) {
                // 移除堆顶元素（即当前最小元素）
                minHeap.poll();
            }
        }
        // 最后剩下的堆顶元素即为数组中第k大的数
        return minHeap.peek();
    }

    // quickselect
    public int findKthLargest(int[] nums, int k) {
        // 先将数组打乱
        shuffle(nums);
        // 用快速选择，递归找到数组中第k大的数
        // 排序好的数组中第k大的数所在的索引位置为len - k
        return findKthLargest(nums, 0, nums.length - 1, nums.length - k);
    }

    private int findKthLargest(int[] nums, int start, int end, int k) {
        // 取范围尾为pivot
        int pivot = nums[end];
        int left = start;
        // 从start开始，将小于pivot的元素移到左边
        for (int i = left; i < end; i++) {
            if (nums[i] < pivot) {
                swap(nums, left++, i);
            }
        }
        // 最后pivot放到了最终的位置上
        swap(nums, left, end);
        // 若刚好pivot就是要找的元素
        if (left == k) {
            return nums[left];
        }
        // 若要找的元素在右边
        if (left < k) {
            return findKthLargest(nums, left + 1, end, k);
        } else {
            // 若要找的元素在左边
            return findKthLargest(nums, start, left - 1, k);
        }
    }

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

    private void shuffle(int[] nums) {
        Random random = new Random();
        for (int i = 1; i < nums.length; i++) {
            int randomIndex = random.nextInt(i + 1);
            swap(nums, i, randomIndex);
        }
    }
}
```

# 102. Binary Tree Level Order Traversal

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

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

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

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

```java
class Solution {
    // iteratively, BFS
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> allRows = new ArrayList<>();
        if (root == null) {
            return allRows;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            int rowSize = queue.size();
            List<Integer> row = new ArrayList<>();
            for (int i = 0; i < rowSize; i++) {
                TreeNode cur = queue.poll();
                row.add(cur.val);
                if (cur.left != null) queue.offer(cur.left);
                if (cur.right != null) queue.offer(cur.right);
            }
            allRows.add(row);
        }
        return allRows;
    }

    // recursively, DFS
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        helper(root, res, 0);
        return res;
    }

    private void helper(TreeNode root, List<List<Integer>> res, int depth) {
        // 若为空树，则返回
        if (root == null) {
            return;
        }
        // 当深度 == res中的元素（list个数）时，说明到了新的一层，需要扩一个位置
        if (depth == res.size()) {
            res.add(new ArrayList<>());
        }
        // NLR的顺序
        // 将根节点中的值加入到与depth对应的list中
        res.get(depth).add(root.val);
        // 遍历左子树
        helper(root.left, res, depth + 1);
        // 遍历右子树
        helper(root.right, res, depth + 1);
    }
}
```

## 289. Game of Life

According to the Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a *board* with *m* by *n* cells, each cell has an initial state *live* (1) or *dead* (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state. The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously.

**Example:**

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

**Follow up**:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

```
class Solution {
    public void gameOfLife(int[][] board) {
        // 用bit的两个位表示两个阶段的状态 00 01 10 11
        int m = board.length, n = board[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                int lives = neighborLives(board, i, j, m, n);
                // 只需要关注什么时候需要置下一个阶段的值为1
                if (board[i][j] == 1 && lives >= 2 && lives <= 3) {
                    board[i][j] = 3;
                }
                if (board[i][j] == 0 && lives == 3) {
                    board[i][j] = 2;
                }
            }
        }
        // 利用右移转换状态到下一个阶段
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] >>= 1;
            }
        }
    }

    // 求出该位置周围的活着的人
    private int neighborLives(int[][] board, int i, int j, int m, int n) {
        int lives = 0;
        // 注意边界
        for (int x = Math.max(0, i - 1); x <= Math.min(m - 1, i + 1); x++) {
            for (int y = Math.max(0, j - 1); y <= Math.min(n - 1, j + 1); y++) {
               lives += board[i][j] & 1;
            }
        }
        // 注意最后要减去自己，因为刚才算了自己的
        lives -= board[i][j] & 1;
        return lives;
    }
}
```

## 378. Kth Smallest Element in a Sorted Matrix

Given a *n* x *n* matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.

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

**Example:**

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

return 13.
```

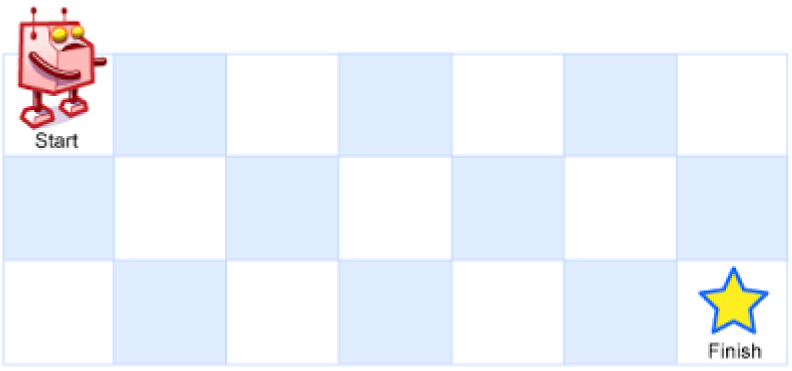
```
class Solution {
    // 时间复杂度O(根号n * logn)
    public int kthSmallest(int[][] matrix, int k) {
        int m = matrix.length, n = matrix[0].length;
        // 最小的数
        int start = matrix[0][0];
        // 最大的数
        int end = matrix[m - 1][n - 1];
        // pari数组中第一个位置存储<=mid的最大的数 第二个位置存储>mid的最小的数
        int[] pair = new int[2];
        while (start < end) {
            // 每次都重置pair
            pair[0] = matrix[0][0];
            pair[1] = matrix[m - 1][n - 1];
            // 求mid
            int mid = start + (end - start) / 2;
            // 找出当前矩阵中<=mid 的数的数量
            int count = lessOrEqual(matrix, pair, mid);
            // 数量少了，往右找（调整值）
            if (count < k) {
                start = pair[1];
            // 数量多了，往左找
            } else if (count > k) {
                end = pair[0];
            // 刚好就是k 返回<=mid的数
            } else {
                return pair[0];
            }
        }
        // 返回start
        return start;
    }

    // 时间复杂度O(根号n)
    // 返回矩阵中<=mid的数的数量
    public int lessOrEqual(int[][] matrix, int[] pair, int mid) {
        int m = matrix.length, n = matrix[0].length;
        // 从左下角开始找
        int row = m - 1;
        int col = 0;
        int count = 0;
        // 循环直到到达边界
        while (row >= 0 && col < n) {
            // 如果当前的数<=mid 那这一列（向上）都小于mid，更新pair[0]，累积count，向右找
            if (matrix[row][col] <= mid) {
                pair[0] = Math.max(pair[0], matrix[row][col]);
                count += row + 1;
                col++;
            } else {
                // 如果当前的数>mid，更新pair[1]，向上找
                pair[1] = Math.min(pair[1], matrix[row][col]);
                row--;
            }
        }
        return count;
    }
}
```

## 62. Unique Paths

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



Above is a 7 x 3 grid. How many possible unique paths are there?

**Example 1:**

```
Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to reach the bottom-right
corner:
1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right
```

**Example 2:**

```
Input: m = 7, n = 3
Output: 28
```

```
class Solution {
    public int uniquePaths(int m, int n) {
        // 一个填表的过程， 状态：dp[i][j]的值表示到达(i,j)的路径有多少条
        int[][] dp = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // 因为是从上到下，从左到有填表，所以边界条件会先得到设置
                if (i == 0 || j == 0) {
                    // 边界条件
                    // 第一行和第一列的路径都只有一条（因为只能往右或者往下）
                    dp[i][j] = 1;
                } else {
                    // 状态转移方程
                    // 其他格子的路径数等于左和上相加
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
                }
            }
        }
        // 返回结果
        return dp[m - 1][n - 1];
    }
}
```

## 454. 4Sum II

Given four lists A, B, C, D of integer values, compute how many tuples `(i, j, k, l)` there are such that `A[i] + B[j] + C[k] + D[l]` is zero.

To make problem a bit easier, all A, B, C, D have same length of N where 0 ≤ N ≤ 500. All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 1.

**Example:**

```
Input:
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

Output:
2

Explanation:
The two tuples are:
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
```

```
class Solution {
    public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
        int count = 0;
        // 建立a + b的和到其频次的映射
        Map<Integer, Integer> map = new HashMap<>();
        for (int a : A) {
            for (int b : B) {
                int sum = a + b;
                // 习惯getOrDefault
                map.put(sum, map.getOrDefault(sum, 0) + 1);
            }
        }
        for (int c : C) {
            for (int d : D) {
                // 等同于做相减操作
                int sum = - c - d;
                // 若有相反的数，则相加
                count += map.getOrDefault(sum, 0);
            }
        }
        return count;
    }
}
```

## 341. Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

**Example 1:**

```
Input: [[1,1],2,[1,1]]
Output: [1,1,2,1,1]
Explanation: By calling next repeatedly until hasNext returns false,
             the order of elements returned by next should be: [1,1,2,1,1].
```

**Example 2:**

```
Input: [1,[4,[6]]]
Output: [1,4,6]
Explanation: By calling next repeatedly until hasNext returns false,
             the order of elements returned by next should be: [1,4,6].
```

```
public class NestedIterator implements Iterator<Integer> {
    // 用一个栈来存储遍历NestedInteger用的迭代器，使得后面来的迭代器先处理
    private Deque<ListIterator<NestedInteger>> iterators;

    public NestedIterator(List<NestedInteger> nestedList) {
        iterators = new LinkedList<>();
        // 将最外层（大）的迭代器压入栈
        iterators.push(nestedList.listIterator());
    }

    @Override
    public Integer next() {
        // 调整位置，保证后面一个元素就是一个Integer
        hasNext();
        // 返回栈顶的迭代器的后面一个元素（同时移动迭代器）
        return iterators.peek().next().getInteger();
    }
    @Override
    public boolean hasNext() {
        // 若已没有迭代器在栈中，说明没有元素待处理
        while (!iterators.isEmpty()) {
            // 如果栈顶的迭代器后面还有待处理的元素
            if (iterators.peek().hasNext()) {
                // 获取元素，向后移动
                NestedInteger cur = iterators.peek().next();
                if (cur.isInteger()) {
                    // 若为整数则回退，并返回true
                    return cur == iterators.peek().previous();
                }
                // 若为list则将其迭代器压入栈，优先处理它
                iterators.push(cur.getList().listIterator());
            } else {
                // 如果迭代器后面没有待处理的元素了，则直接将其弹出栈
                iterators.pop();
            }
        }
        return false;
    }
}
```

# 384. Shuffle an Array

Shuffle a set of numbers without duplicates.

**Example:**

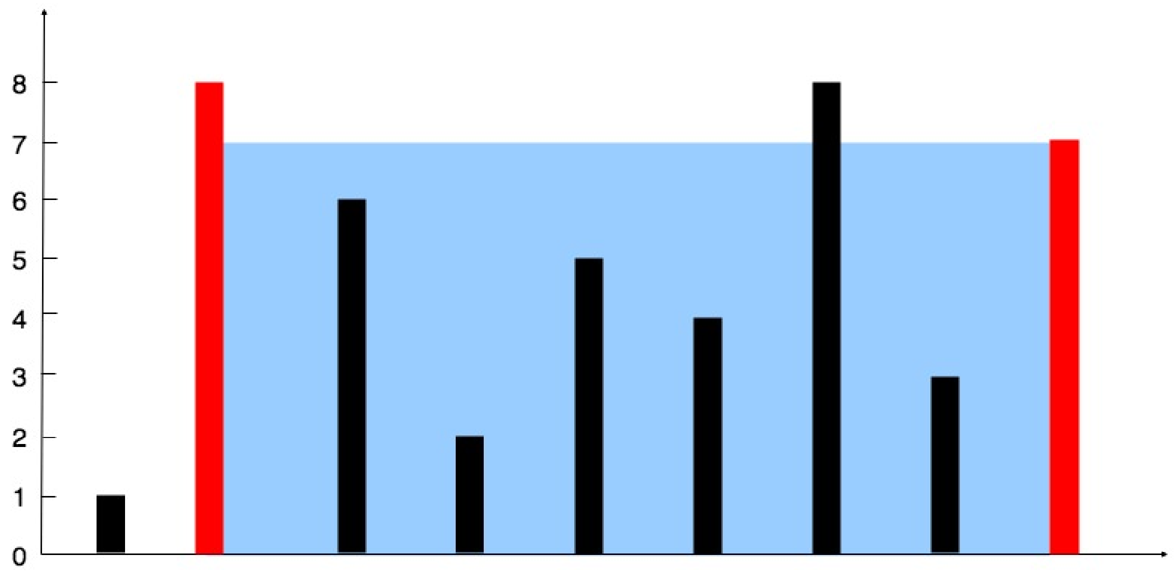
```
// Init an array with set 1, 2, and 3.
int[] nums = {1,2,3};
Solution solution = new Solution(nums);

// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3]
must equally likely to be returned.
solution.shuffle();

// Resets the array back to its original configuration [1,2,3].
solution.reset();

// Returns the random shuffling of array [1,2,3].
solution.shuffle();
```

```
class Solution {
    // 存储最初的数组
    private int[] nums;
    // 用于随机产生索引
    private Random random;

    public Solution(int[] nums) {
        this.nums = nums;
        random = new Random();
    }
    public int[] reset() {
        return nums;
    }

    public int[] shuffle() {
        // 利用clone()方法进行复制（因为nums里面存储的是基本数据类型，所以浅复制就OK）
        int[] copy = nums.clone();
        for (int i = 1; i < copy.length; i++) {
            // 随机产生[0, i]的整数
            int r = random.nextInt(i + 1);
            // 交换
            swap(copy, r, i);
        }
        return copy;
    }

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

## 11. Container With Most Water

Given *n* non-negative integers *a1*, *a2*, ..., *an* , where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

**Note:** You may not slant the container and *n* is at least 2.



The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example:**

```
Input: [1,8,6,2,5,4,8,3,7]
Output: 49
```

```
class Solution {
    /*
    1. The widest container (using first and last line) is a good candidate,
because        of its width. Its water level is the height of the smaller one of
first and         last line.
    2. All other containers are less wide and thus would need a higher water
level in       order to hold more water.
    3. The smaller one of first and last line doesn't support a higher water
level and  can thus be safely removed from further consideration.
    */
    public int maxArea(int[] height) {
        int area = 0;
        int i = 0, j = height.length - 1;
        while (i < j) {
            area = Math.max(area, (j - i) * Math.min(height[i], height[j]));
            if (height[i] < height[j]) {
                i++;
            } else {
                j--;
            }
        }
        return area;
    }
}
```

## 208. Implement Trie (Prefix Tree)

Medium

321750Add to ListShare

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.

```
// 前缀树，可以理解为26叉树
```

```
class Trie {
    // 根结点
    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String s) {
        // 当前引用
        TrieNode curr = root;
        // 遍历输入的字符串
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            // 如果没有当前字母对应的孩子结点，就创建一个
            if (curr.letters[c - 'a'] == null) {
                curr.letters[c - 'a'] = new TrieNode();
            }
            // 移动引用：实质上跟curr = curr.left差不多
            curr = curr.letters[c - 'a'];
        }
        // 标记在字典树中存在这么一个词
        curr.isWord = true;
    }

    public boolean search(String word) {
        // 当前引用
        TrieNode curr = root;
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            // 如果不存在这个字母
            if (curr.letters[c - 'a'] == null) {
                return false;
            }
            curr = curr.letters[c - 'a'];
        }
        // 存在这个词
        return curr.isWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode curr = root;
        for (int i = 0; i < prefix.length(); i++) {
            char c = prefix.charAt(i);
            if (curr.letters[c - 'a'] == null) {
                return false;
            }
            curr = curr.letters[c - 'a'];
        }
        // 能遍历完即可
        return true;
    }
    // 前缀树结点，全部小写就是26叉树的结点
    private class TrieNode {
        // 标记当前遍历完的字符串是不是一个存在的词
        boolean isWord;
        // 存了26个指向子节点的引用（二叉树是left, right）
        // letters[0]不为空：代表着当前字母为a
        TrieNode[] letters = new TrieNode[26];
    }
}
```

## 36. Valid Sudoku

Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the 9 `3x3` sub-boxes of the grid must contain the digits `1-9` without repetition.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

**Example 2:**

```
Input:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner being
    modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is
invalid
```

```
class Solution {
    public boolean isValidSudoku(int[][] board) {
        // 利用set添加重复元素时会返回false的特性
        Set<String> seen = new HashSet<>();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                int num = board[i][j];
                // 如果当前元素时字母
                if (num != '.') {
                    // 如果当前元素同行、同列或者同块重复，返回false
                    if (!seen.add(num + " in row " + i) ||
                        !seen.add(num + " in col " + j) ||
                        !seen.add(num + " in block " + i/3 + "-" + j/3)) {
                         return false;
                    }
                }
            }
        }
        return true;
    }
}
```

## 105. Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

**Note:**
You may assume that duplicates do not exist in the tree.

For example, given

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Return the following binary tree:

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

```
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        // 借助一个辅助方法，递归构造二叉树
        return helper(preorder, inorder, 0, 0, inorder.length - 1);
    }
    // 注意preStart, inStart, inEnd的取值范围皆为[0, len - 1]
    private TreeNode helper(int[] preorder, int[] inorder, int preStart, int
inStart, int inEnd) {
        // 如果先序序列为空， 或者中序序列为空，返回一个空结点
        if (preStart > preorder.length - 1 || inStart > inEnd) {
            return null;
        }
        // 构造当前子树的根结点
        TreeNode root = new TreeNode(preorder[preStart]);
        int index = 0;
        // 在中序遍历序列中找到根结点的索引，根据索引就可把左右子树区分开来
        for (int i = 0; i < inorder.length; i++) {
            if (inorder[i] == preorder[preStart]) {
                index = i;
                break;
            }
        }
        // 递归构造左子树，并返回左子树的根结点（传入左子树的先序，和中序序列的索引）
        root.left = helper(preorder, inorder, preStart + 1, inStart, index - 1);
        // 递归构造右子树，并返回右子树的根结点（传入右子树的先序和中序的索引）
        root.right = helper(preorder, inorder, preStart + index - inStart + 1,
index + 1, inEnd);
        // 返回构造好的根结点
        return root;
    }
}
```

| | left | right |
|---|---|---|
| preStart | preStart + 1 | preStart + (index - inStart)左子树结点个数 + 1 |
| inStart | inStart | index + 1 |
| inEnd | index - 1 | inEnd |

# 380. Insert Delete GetRandom O(1)

Design a data structure that supports all following operations in *average* **O(1)** time.

1. `insert(val)`: Inserts an item val to the set if not already present.
2. `remove(val)`: Removes an item val from the set if present.
3. `getRandom`: Returns a random element from current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

**Example:**

```
// Init an empty set.
RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.
randomSet.insert(1);

// Returns false as 2 does not exist in the set.
randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].
randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.
randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].
randomSet.remove(1);

// 2 was already in the set, so return false.
randomSet.insert(2);
```

```
// Since 2 is the only number in the set, getRandom always return 2.
randomSet.getRandom();
```

```java
class RandomizedSet {
    Random random;
    // 用ArrayList装数据
    List<Integer> nums;
    // 用map存储数据到其在数组中对应的下标的映射
    Map<Integer, Integer> numToIndex;

    public RandomizedSet() {
        random = new Random();
        nums = new ArrayList<>();
        numToIndex = new HashMap<>();
    }

    public boolean insert(int val) {
        // 利用map的特性（查找为O(1)），判断是否已存在该元素
        if (numToIndex.containsKey(val)) {
            return false;
        }
        // 添加映射关系
        numToIndex.put(val, nums.size());
        // 把数据添加到数组尾部
        nums.add(val);
        return true;
    }

    public boolean remove(int val) {
        if (!numToIndex.containsKey(val)) {
            return false;
        }
        // 获取所删除的元素在数组中的下标
        int index = numToIndex.get(val);
        // 如果删除的元素不在数组尾
        if (index < nums.size() - 1) {
            // 用数组尾的元素覆盖掉它
            int tail = nums.get(nums.size() - 1);
            nums.set(index, tail);
            // 同时更新尾元素在map中的索引信息
            numToIndex.put(tail, index);
        }
        // 保证了删除的永远是数组的最后一个元素O(1)
        nums.remove(nums.size() - 1);
        // 删除键值对O(1)
        numToIndex.remove(val);
        return true;
    }

    public int getRandom() {
        // 利用nextInt随机返回一个数组中的数（利用了数组的随机存取特性）
        return nums.get(random.nextInt(nums.size()));
    }
}
```

# 75. Sort Colors

Given an array with *n* objects colored red, white or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

**Note:** You are not suppose to use the library's sort function for this problem.

**Example:**

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

**Follow up:**

- A rather straight forward solution is a two-pass algorithm using counting sort.
  First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
- Could you come up with a one-pass algorithm using only constant space?

```
class Solution {
    // counting sort
    public void sortColors(int[] nums) {
        int[] count = new int[3];
        // 遍历原数组，统计0,1,2的出现次数
        for (int n : nums) {
            count[n]++;
        }
        // 覆盖的写入点
        int insertPos = 0;
        // 遍历计数数组，根据出现次数，写值
        for (int i = 0; i < count.length; i++) {
            while (count[i] > 0) {
                nums[insertPos++] = i;
                count[i]--;
            }
        }
    }
    // so-called one-pass solution
    public void sortColors(int[] nums) {
        // 三个指针，刚开始位置为-1
        // k负责记录0 1 2，同时写入2
        // j负责记录0 1， 同时写入1
        // i负责记录0， 同时写入0
        int i = -1, j = -1, k = -1;
        // 用测试数据[0,1,2]去想就好了
        for (int p = 0; p < nums.length; p++) {
            if (nums[p] == 0) {
                nums[++k] = 2;
                nums[++j] = 1;
                nums[++i] = 0;
            } else if (nums[p] == 1) {
                nums[++k] = 2;
                nums[++j] = 1;
            } else if (nums[p] == 2) {
                nums[++k] = 2;
            }
        }
    }
}
```

## 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.
```

```
class Solution {
    public int numSquares(int n) {
        // dp[i]表示数字i所需的最少的完全平方数（想想递归怎么做：dp可以避免重复计算子问题）
        int[] dp = new int[n + 1];
        // 遍历dp数组，填表
        for (int i = 1; i <= n; i++) {
            // 先设dp[i]的值为最大，方便后面取较小值
            dp[i] = Integer.MAX_VALUE;
            // 遍历<=当前数的完全平方数
            for (int j = 1; j * j <= i; j++) {
                // 当前数的最小是 dp[i - j * j] + 1(这个1就是一个j * j（1 4 9..）)
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }
        // 返回所求的值
        return dp[n];
    }
}
```

# 103. Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

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

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

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

```
class Solution {
    // iteratively BFS
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        // 存储所有行的元素
        List<List<Integer>> rows = new ArrayList<>();
        if (root == null) {
            return rows;
        }
        // 存储结点的队列，用于实现层级遍历
        Queue<TreeNode> queue = new LinkedList<>();
        // 根结点入队
        queue.offer(root);
        // 控制是头插法还是尾插法
        boolean normalOrder = true;
        while (!queue.isEmpty()) {
            // 获取当前层的元素的个数（也就是队列的大小）
            int rowSize = queue.size();
            // 存储当前层的元素，一定要用链表，因为在头部插入是O(1)
            List<Integer> row = new LinkedList<>();
            // 遍历当前层的元素
            for (int i = 0; i < rowSize; i++) {
                TreeNode curr = queue.poll();
                // 正常顺序是尾插
                if (normalOrder) {
                    row.add(curr.val);
                } else {
                    // 非正常顺序是头插法
                    row.add(0, curr.val);
                }
```

```java
                if (curr.left != null) queue.offer(curr.left);
                if (curr.right != null) queue.offer(curr.right);
            }
            // 将当前层的元素入队
            rows.add(row);
            // 改变插入方式
            normalOrder = !normalOrder;
        }
        return rows;
    }

    // recursively
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> rows = new ArrayList<>();
        dfs(rows, root, 0);
        return rows;
    }

    private void dfs(List<List<Integer>> rows, TreeNode root, int depth) {
        // 递归边界，如果当前结点为null，则返回
        if (root == null) {
            return;
        }
        // rows是随着depth变的，没当到达一个新的层，就为这个层创建一个链表
        if (rows.size() <= depth) {
            rows.add(new LinkedList<>());
        }
        // 利用depth取到当前层的list的引用，利用的随机存取特性，所以外面必须用ArrayList
        List<Integer> currentRow = rows.get(depth);
        // 根据深度判断头插还是尾插
        if (depth % 2 == 0) {
            currentRow.add(root.val);
        } else {
            currentRow.add(0, root.val);
        }
        // 递归遍历左子树
        dfs(rows, root.left, depth + 1);
        // 递归遍历右子树
        dfs(rows, root.right, depth + 1);
    }
}
```

# 131. Palindrome Partitioning

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of *s*.

**Example:**

```
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

```
class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> res = new ArrayList<>();
        if (s == null || s.length()) {
            return res;
        }
        // dp数组dp[left][right] == true 表示 left->right为一个回文串
        boolean[][] dp = new boolean[s.length()][s.length()];
        for (int i = 0; i < s.length(); i++) {
            for (int j = 0; j <= i; j++) {
                // 若两个字符相等，且它们的相距不超过2 或者是它们里面也是回文串（递归）
                // 则它们是回文串
                if (s.charAt(j) == s.charAt(i) && ((i - j <= 2) || dp[j + 1][i -
1])) {
                    dp[j][i] = true;
                }
            }
        }
        dfs(res,dp, new ArrayList<>(), s, 0);
        return res;
    }

    private void dfs(List<List<String>> res, boolean[][] dp, List<String>
tempList, String s, int left) {
        // left到达字符串长度，说明已经到叶子结点了，已经没有数据了，加入数据，返回上层调用的
地方
        if (left == s.length()) {
            res.add(new ArrayList<>(tempList));
            return;
        }
        // i = left
        for (int i = left; i < s.length(); i++) {
            // 若子串left-i 为回文串
            if (dp[left][i]) {
                // 将它们加入tempList中
                tempList.add(s.substring(left, i + 1));
                // 递归探索 传参: left = i + 1
                dfs(res, dp, tempList, s, i + 1);
                // 返回， 删除tempList中最后一个元素，为探索新的组合
                tempList.remove(tempList.size() - 1);
            }
        }
    }
}
```

# 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 2:**

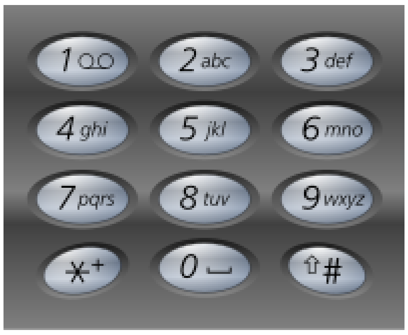
```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

```
class Solution {
    private int row;
    private int col;

    public int numIsIsland(char[][] grid) {
        if (grid == null || grid.length == 0) {
            return 0;
        }
        row = grid.length;
        col = grid[0].length;
        int count = 0;
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                // 如果遇到了地面
                if (grid[i][j] == '1') {
                    // 将这块岛屿沉了
                    sink(grid, i, j);
                    count++;
                }
            }
        }
        return count;
    }

    private void sink(int[][] grid, int i, int j) {
        // 递归边界条件
        if (grid[i][j] == '0' || i < 0 || j < 0 || i >= row || j >= col) {
            return;
        }
        // 沉没当前地面
        grid[i][j] = '0';
        // 递归沉没上下左右的地面
        sink(grid, i - 1, j);
        sink(grid, i + 1, j);
        sink(grid, i, j - 1);
        sink(grid, i, j + 1);
        // 方法返回时该岛屿已经沉没
    }
}
```

# 17. Letter Combinations of a Phone Numbe

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



**Example:**

```
Input: "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

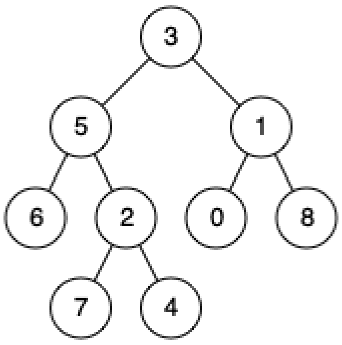
```
class Solution {
    public List<String> letterCombinations(String digits) {
        // LinkedList实现了Deque, Queue, List接口，所以有他们所有方法的实现
        LinkedList<String> res = new LinkedList<>();
        if (digits == null || digits.length() == 0) {
            return res;
        }
        // int -> String的映射可以直接用一个String[]来表示
        String[] map = {"0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs",
"tuv", "wxyz"};
        // 填入一个空字符串，来统一操作
        res.offer("");
        // 遍历数字字符串
        for (int i = 0; i < digits.length(); i++) {
            // 获取当前遍历到的数 FIFO
            int num = digits.charAt(i) - '0';
            // 点睛之笔：用队列头的字符串的长度，来判断是否要进入下一个数的迭代
            while (res.peek().length() == i) {
                // 取出队头元素
                String s = res.poll();
                // 将队头元素与num对应的字符串中的字符进行拼接，然后再加入到队列中
                for (char c : map[num].toCharArray()) {
                    res.offer(s + c);
                }
            }
        }
        return res;
    }
}
```

# 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.
```

```
class Solution {
    // recursively
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        // 递归出口：如果root为null，或者root就是p，q中的一个，则返回root
        if (root == null || root == p || root == q) {
            return root;
        }
        // 递归在左子树中找到p，q的最低公共祖先或者它们自身
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        // 递归在右子树中找到p，q的最低公共祖先或者它们自身
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        // 如果左子树中没有找到，则返回右子树中的结果
        if (left == null) {
            return right;
        }
        // 如果右子树中没有找到，则返回左子树中的结果
        if (right == null) {
            return left;
        }
        // 如果两个树中都找到了（一个为p，一个为q），说明当前结点就是它们的最小根结点，返回它
        return root;
    }

    // iteratively
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        // 判断特例
        if (root == null || root == p || root == q) {
            return root;
        }
        // 通过map，将子节点与其对应的父结点连接起来
        Map<TreeNode, TreeNode> childToParent = new HashMap<>();
        // 用于非递归遍历树
        Deque<TreeNode> stack = new LinkedList<>();
        // 根结点没有父节点
        childToParent.put(root, null);
        // 根节点入栈
        stack.push(root)
        // 遍历直到p和q都能找到，利用了hashmap查找O(1)的特性，注意要用||
        while (!childToParent.containsKey(p) || !childToParent.containsKey(q)) {
            // 弹出元素
            TreeNode curr = stack.pop()
            if (curr.right != null) {
                // 在原来的操作之上加了记录节点的父节点的操作
                childToParent.put(curr.right, curr);
                stack.push(curr.right);
            }
            if (curr.left != null) {
                childToParent.put(curr.left, curr);
                stack.push(curr.left);
            }
        }
        // 将其中的一个目标节点的祖先（祖先也包括自己），都加入一个set中
        Set<TreeNode> ancestors = new HashSet<>();
        while (p != null) {
            ancestors.add(p);
            p = childToParent.get(p);
        }
        // 从另一个目标节点出发，碰到set的第一个节点就是它们的最低公共祖先
        // 利用了hashset查找O(1)的特性
        while (!ancestors.contains(q)) {
            q = childToParent.get(q);
        }
        return q;
    }
}
```

# 116. Populating Next Right Pointers in Each Node

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

**Follow up:**

- You may only use constant extra space.
- Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

**Example 1:**

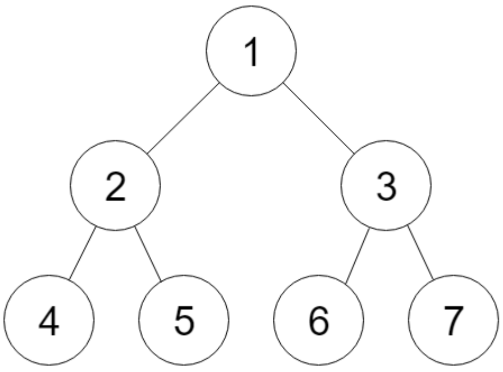


Figure A

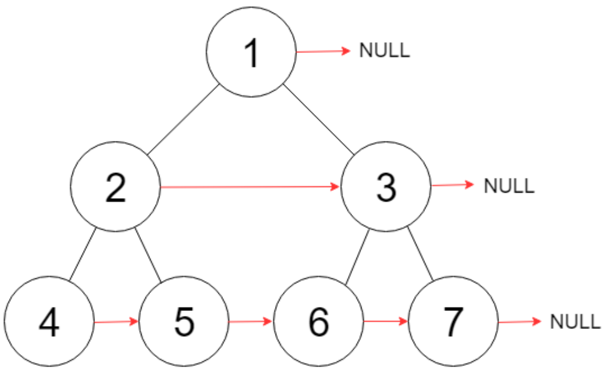


Figure B

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

```
class Solution {
    public Node connect(Node root) {
        // 判断特例
        if (root == null) {
            return null;
        }
        // 沿着最左分支向下遍历
        Node prev = root;
        // 当还没到最底层时
        while (prev.left != null) {
            // curr指向当前层的的第一个节点
            Node curr = prev;
            // 遍历当前层
            while (curr != null) {
                // 将自己的左孩子和右孩子连接起来
                curr.left.next = curr.right;
                // 如果右边还有节点，就将自己的右孩子和右边节点的左孩子连接起来
                if (curr.next != null) {
                    curr.right.next = curr.next.left;
                }
                // 向后遍历
                curr = curr.next;
            }
            // 向下遍历
            prev = prev.left;
        }
        return root;
    }
}
```

# 162. Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array `nums`, where `nums[i] ≠ nums[i+1]`, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that `nums[-1] = nums[n] = -∞`.

**Example 1:**

```
Input: nums = [1,2,3,1]
Output: 2
Explanation: 3 is a peak element and your function should return the index number
2.
```

**Example 2:**

```
Input: nums = [1,2,1,3,5,6,4]
Output: 1 or 5
Explanation: Your function can return either index number 1 where the peak
element is 2,
             or index number 5 where the peak element is 6.
```

**Follow up:** Your solution should be in logarithmic complexity.

```
class Solution {
    // iteratively
    public int findPeakElement(int[] nums) {
        // 二分查找
        int low = 0, high = nums.length - 1;
        // 当low -> <- high 相等时，退出循环
        while (low < high) {
            // 中点
            int mid1 = low + (high - low) / 2;
            // 中点 + 1
            int mid2 = mid1 + 1;
            // 因为low和high最后要合在一起
            // 所以low的左边应该小于它
            // high的右边应该小于它，这样才能保证是个峰值
            if (nums[mid1] < nums[mid2]) {
                low = mid2;
            } else {
                high = mid1;
            }
        }
        return low;
    }

    // recursively
    public int findPeakElement(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }

    private int helper(int[] nums, int low, int high) {
        // 递归出口: left == right
        if (low == high) {
            return low;
        }
        int mid1 = low + (high - low) / 2;
        int mid2 = mid1 + 1;
        if (nums[mid1] < nums[mid2]) {
            return helper(nums, mid2, high);
        } else {
            return helper(nums, low, mid1);
        }
    }
}
```

# 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`.

```
class Solution {
    // time complexity O(m + n)
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0) {
            return false;
        }
        int row = matrix.length;
        int col = matrix[0].length;
        int i = row - 1;
        int j = 0;
        // 当没有到达边界
        while (i >= 0 && j < col) {
            // 剪枝的思想，认真体会
            // 如果当前元素小于target，则剪掉该列上面的元素(均小于它)，往右边走
            if (matrix[i][j] < target) {
                j++;
                // 如果当前元素大于target，则剪掉该行右边的元素（均大于它），往上面走
            } else if (matrix[i][j] > target) {
                i--;
            } else {
                return true;
            }
        }
        return false;
    }
}
```

# 73. Set Matrix Zeroes

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in-place.

**Example 1:**

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

**Example 2:**

```
Input:
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
Output:
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]
```

**Follow up:**

- A straight forward solution using O(*m***n*) space is probably a bad idea.
- A simple improvement uses O(*m* + *n*) space, but still not the best solution.
- Could you devise a constant space solution?

```
class Solution {
    public void setZeroes(int[][] matrix) {
        // 总体思想：利用第一行和第一列做标记
        // 判断特殊条件
        if (matrix == null || matrix.length == 0) {
            return;
        }
        // 标记第一行和第一列有没有0，即它们需不需要全置为0
        boolean firstRow = false, firstCol = false;
        int m = matrix.length, n = matrix[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // 如果当前元素为0
                if (matrix[i][j] == 0) {
                    // 做标记，该行元素置0
                    matrix[i][0] = 0;
                    // 做标记，该列元素置0
                    matrix[0][j] = 0;
                    // 0元素出现在第一行
                    if (i == 0) {
                        firstRow = true;
                    }
                    // 0元素出现在第一列
                    if (j == 0) {
                        firstCol = true;
                    }
                }
            }
        }
        // 处理除了第一行和第一列外的元素
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
        // 处理第一列
        if (firstCol) {
            for (int i = 0; i < m; i++) {
                matrix[i][0] = 0;
            }
        }
        // 处理第一行
        if (firstRow) {
            for (int j = 0; j < n; j++) {
                matrix[0][j] = 0;
            }
        }
    }
}
```

## 207. Course Schedule

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

**Example 1:**

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is
possible.
```

**Example 2:**

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take
course 0 you should
             also have finished course 1. So it is impossible.
```

```
class Solution {
    // 拓扑排序
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // 用邻接表存储图
        List<List<Integer>> adjs = new ArrayList<>();
        // 记录入度
        int[] indegree = new int[numCourses];
        // 初始化图和入度
        init(adjs, indegree, prerequisites);
        // 队列保存即将要访问的结点，即入度为0的结点
```

```java
        Queue<Integer> toVisit = new LinkedList<>();
        for (int i = 0; i < indegree.length; i++) {
            if (indegree[i] == 0) {
                toVisit.offer(i);
            }
        }
        // 已经访问过的结点
        int visited = 0;
        // 拓扑排序序列
        int[] order = new int[numCourses];
        // 当图中还有入度为0的结点
        while (!toVisit.isEmpty()) {
            // 取出该元素
            int from = toVisit.poll();
            // 如果不需要序列，可以直接visited++
            order[visited++] = from;
            // 删除入度为0的结点后，它后面的相邻结点入度-1
            for (int to : adjs.get(from)) {
                indegree[to]--;
                // 若-1后该结点入度为0，将该结点加入待访问的队列
                if (indegree[to] == 0) {
                    toVisit.offer(to);
                }
            }
        }
        return visited == numCourses;
    }

    private void init(List<List<Integer>> adjs, int[] indegree, int[][]
prerequisites) {
        // 初始化领接表
        for (int i = 0; i < indegree.length; i++) {
            adjs.add(new LinkedList<>());
        }
        // 初始化领接表和入度数组
        for (int[] edge : prerequisites) {
            adjs.get(edge[1]).add(edge[0]);
            indegree[edge[0]]++;
        }
    }
}
```

## 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?

Our strategy determined by the following conditions,

```
1. If A[i] is smallest among all end
   candidates of active lists, we will start
   new active list of length 1.
2. If A[i] is largest among all end candidates of
  active lists, we will clone the largest active
  list, and extend it by A[i].
3. If A[i] is in between, we will find a list with
  largest end element that is smaller than A[i].
  Clone and extend this list by A[i]. We will discard all
  other lists of same length as that of this modified list.
```

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        // 特殊处理
        if (nums == null || nums.length == 0) {
            return 0;
        }
        // 存放所有递增“序列”的尾元素
        int[] tailTable = new int[nums.length];
        // 遇到第一个元素，创建一个新的序列
        tailTable[0] = nums[0];
        // len一直指向空的地方
        int len = 1;
        // 遍历后边的元素
        for (int i = 1; i < nums.length; i++) {
            // new smallest value
            // 如果当前元素小于首个尾元素，替换掉
            if (nums[i] < tailTable[0]) {
                tailTable[0] = nums[i];
            // nums[i] wants to extend largest subsequence
            // 如果当前元素比最后一个尾元素要大，直接添加
            } else if (nums[i] > tailTable[len - 1]) {
                tailTable[len++] = nums[i];
            } else {
                // nums[i] wants to be current end candidate of an existing
                // subsequence. It will replace ceil value in tailTable
                // 如果处理中间位置，则用二分查找在递增序列中找到第一个>=它的元素的位置
                // 并替换掉到那个位置的元素
                int index  =  ceilIndex(tailTable, -1, len - 1, nums[i]);
                tailTable[index] = nums[i];
            }
        }
        return len;
    }

    // 二分查找在递增序列中找到第一个>=target的元素的位置
    private int ceilIndex(int[] nums, int left, int right, int target) {
        while (right - left > 1) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid;
            } else {
                right = mid;
            }
        }
        return right;
    }
}
```

## 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
```

```
class Solution {
    // 归并排序
    public ListNode sortList(ListNode head) {
        // 哑结点
        ListNode dummy = new ListNode(0);
        // 记录头结点
        dummy.next = head;
        // 计算链表长度
        int len = 0;
        while (head != null) {
            head = head.next;
            len++;
        }
        // 如果链表长度为4 则只需2次归并排序 所以step=1 2 就行
        for (int step = 1; step < len; step <<= 1) {
            // 每次都重新让prev 和 curr指向链表头部
            ListNode prev = dummy;
            ListNode curr = dummy.next;
            // 循环直至curr为空，说明当前step的排序已经处理完了
            while (curr != null) {
                // left指向左边的有序序列
                ListNode left = curr;
                // right指向右边的有序序列
                ListNode right= split(left, step);
                // curr指向下一个要处理的地方
                curr = split(right, step);
                // merge将left, right指向的两个有序序列合并，并将尾结点返回
                prev = merge(left, right, prev);
            }
        }
        // 返回头结点
        return dummy.next;
    }

    private ListNode split(ListNode head, int step) {
        // 当链表个数为奇数时，会出现传入的ListNode为null的情况
        if (head == null) {
            return null;
        }
        // 找到当前步长的尾
        for (int i = 1; i < step && head.next != null; i++) {
            head = head.next;
        }
        // 记录右边
        ListNode right = head.next;
        // 断开
        head.next = null;
        // 返回右边
        return right;
    }
    // 简单的合并两个有序序列，并将尾结点返回
    private ListNode merge(ListNode left, ListNode right, ListNode prev) {
        ListNode curr = prev;
        while (left != null && right != null) {
            if (left.val < right.val) {
                curr.next = left;
                left = left.next;
            } else {
                curr.next = right;
                right = right.next;
            }
            curr = curr.next;
        }
        if (left != null) {
            curr.next = left;
        }
        if (right != null) {
            curr.next = right;
        }
        while (curr.next != null) {
            curr = curr.next;
        }
        return curr;
    }
}
```

# 210. Course Schedule II

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

**Example 1:**

```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should
have finished
             course 0. So the correct course order is [0,1] .
```

**Example 2:**

```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should
have finished both
             courses 1 and 2. Both courses 1 and 2 should be taken after you
finished course 0.
             So one correct course order is [0,1,2,3]. Another correct ordering
is [0,2,1,3] .
```

```
class Solution {
    // 拓扑排序
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        // 邻接表
        List<List<Integer>> adjs = new ArrayList<>();
        // 入度
        int[] indegree = new int[numCourses];
        // 待访问的数组
        Queue<Integer> toVisit = new LinkedList<>();
        // 初始化
        init(adjs, indegree, toVisit, prerequisites);

        int[] order = new int[numCourses];
        int visited = 0;
        while (!toVisit.isEmpty()) {
            int from = toVisit.poll();
            order[visited++] = from;
            for (int to : adjs.get(from)) {
                indegree[to]--;
                if (indegree[to] == 0) {
                    toVisit.offer(to);
                }
            }
        }
        return visited == numCourses ? order : new int[0];
    }

    private void init(List<List<Integer>> adjs, int[] indegree, Queue<Integer>
toVisit, int[][] prerequisites) {
        for (int i = 0; i < indegree.length; i++) {
            adjs.add(new LinkedList<>());
        }
        for (int[] edge : prerequisites) {
            adjs.get(edge[1]).add(edge[0]);
            indegree[edge[0]]++;
        }
        for (int i = 0; i < indegree.length; i++) {
            if (indegree[i] == 0) {
                toVisit.offer(i);
            }
        }
    }
}
```

# 139. Word Break

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

**Note:**

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

**Example 1:**

```
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

**Example 2:**

```
Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen
apple".
             Note that you are allowed to reuse a dictionary word.
```

**Example 3:**

```
Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
Output: false
```

```
class Solution {
    public boolean wordbreak(String s, List<String> wordDict) {
        // dp[i]表示[0, i)这部分字符串可分
        boolean[] dp = new boolean[s.length() + 1];
        // [0, 0)为空串，默认可分
        dp[0] = true;
        // 遍历字符串
        for (int i = 1; i <= s.length(); i++) {
            // 从头开始找
            for (int j = 0; j < i; j++) {
                // 若[0, j)可分，且[j, i)也在字符串数组中
                if (dp[j] && wordDick.contains(s.substring(j, i))) {
                    // 则[0, i)可分
                    dp[i] =  true;
                    // 已经判断完了，跳出内存循环
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```

## 334. Increasing Triplet Subsequence

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Formally the function should:

> Return true if there exists *i, j, k*
> such that *arr[i]* < *arr[j]* < *arr[k]* given 0 ≤ *i* < *j* < *k* ≤ *n*-1 else return false.

**Note:** Your algorithm should run in O(*n*) time complexity and O(*1*) space complexity.

**Example 1:**

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

**Example 2:**

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

```
class Solution {
    public boolean increasingTriplet(int[] nums) {
        if (nums == null || nums.length == 0) {
            return false;
        }
        int min = Integer.MAX_VALUE, secondMin = Integer.MAX_VALUE;
        for (int n : nums) {
            // 更新最小值 这里需用<= 想想1 1 1的情况
            if (n <= min) {
                min = n;
            } else if (n <= secondMin) {
                // 更新第二小的值
                secondMin = n;
            } else {
                // 若能走到这，说明nums[i] < nums[j] < num[k] 已成立 i < j < k
                // 而当前这个就是nums[k]
                return true;
            }
        }
        return false;
    }
}
```

## 56. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

**Example 1:**

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

**Example 2:**

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

```
class Solution {
    public int[][] merge(int[][] intervals) {
        // 为null，或者长度<=1，直接返回
        if (intervals == null || intervals.length <= 1) {
            return intervals;
        }
        // 当输入用例为[[1,4],[0,4]]时，需要重新根据第一个元素排序，要不然结果是[1,4]结果错
误
        // sort第二个参数是comparator的实现子类对象，只要重写compare方法就行
        Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0], i2[0]));
        // 用list存储int[] int[]也是一个Object
        List<int[]> res = new ArrayList<>();
        // 将首个元素放入list
        res.add(intervals[0]);
        // tail指向list中的尾元素
        int[] tail = intervals[0];
        for (int[] interval : intervals) {
            // 遍历intervals，如果当前元素的左边界<= 尾元素的右边界
            // 那就尝试去更新尾元素的右边界
            if (interval[0] <= tail[1]) {
                tail[1] = Math.max(tail[1], interval[1]);
            } else {
                // 如果当前元素的左边界 > 尾元素的右边界，则直接添加
                // 并使tail指向新的尾元素
                res.add(interval);
                tail = interval;
            }
        }
        return res.toArray(new int[res.size()][]);
        // 指定泛型类型T，将集合中的元素转化为T[]
        //<T> T[] toArray(T[] a)  此时这里的T类型擦除后就是Object
        // 所以用int[]替换T的位置，就是传一个 int[][]进去即可，且可以指定数组的对象
    }
}
```

# 134. Gas Station

There are *N* gas stations along a circular route, where the amount of gas at station *i* is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

**Note:**

- If there exists a solution, it is guaranteed to be unique.
- Both input arrays are non-empty and have the same length.
- Each element in the input arrays is a non-negative integer.

**Example 1:**

```
Input:
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]

Output: 3

Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 =
4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to
station 3.
Therefore, return 3 as the starting index.
```

**Example 2:**

```
Input:
gas  = [2,3,4]
cost = [3,4,3]

Output: -1

Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the
next station.
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only
have 3.
Therefore, you can't travel around the circuit once no matter where you start.
```

```java
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        // 出发点
        int start = 0;
        // 油的总数
        int gasSum = 0;
        // 耗油总数
        int costSum = 0;
        // 当前车子的油箱
        int tank = 0;
        for (int i = 0; i < gas.length; i++) {
            // 累积两个
            gasSum += gas[i];
            costSum += cost[i];
            // 计算当前油箱中的油数
            tank += gas[i] - cost[i];
            // 如果当前油箱为负，说明之前的点都无法作为开始点，因为到达不了
            // 所以tank清0，start指向下一个点
            if (tank < 0) {
                start = i + 1;
                tank = 0;
            }
        }
        // 如果油的总数>=耗油的总数，那一定存在出发点
        if (gasSum >= costSum) {
            return start;
        }
        return -1;
    }
}
```

# 227. Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only **non-negative** integers, `+`, `-`, `*`, `/` operators and empty spaces ``. The integer division should truncate toward zero.

**Example 1:**

```
Input: "3+2*2"
Output: 7
```

**Example 2:**

```
Input: " 3/2 "
Output: 1
```

**Example 3:**

```
Input: " 3+5 / 2 "
Output: 5
```

```
class Solution {
    public int calculate(String s) {
        // 判特殊条件
        if (s == null || s.length() == 0) {
            return 0;
        }
        // 在最后再添加一个操作符作为最后一个操作符的触发器
        s += '+';
        // num初值为0
        int num = 0;
        // op初值为+，即将第一个元素压入栈
        char op = '+';
        // 操作数栈
        Deque<Integer> stack = new LinkedList<>();
        for (char c : s.toCharArray()) {
            // 如果当前遍历到的是数字，则计算其值
            if (c >= '0' && c <= '9') {
                num = num * 10 + c - '0';
                continue;
            }
            // 遇到空字符，跳过
            if (c == ' ') {
                continue;
            }
            // 后面的均为遇到操作符：遇到操作符，就触发前面一个操作符
            if (op == '+') {
                stack.push(num);
            }
            if (op == '-') {
                stack.push(-num);
            }
            if (op == '*') {
                stack.push(stack.pop() * num);
            }
            if (op == '/') {
                stack.push(stack.pop() / num);
            }
            // 使用前面的一个操作符计算完之后将num置0，更新操作符
            num = 0;
            op = c;
        }
        // 累计栈中的元素
        int total = 0;
        while (!stack.isEmpty()) {
            total += stack.pop();
        }
        return total;
    }
}
```

# 138. Copy List with Random Pointer

Medium

3451681Add to ListShare

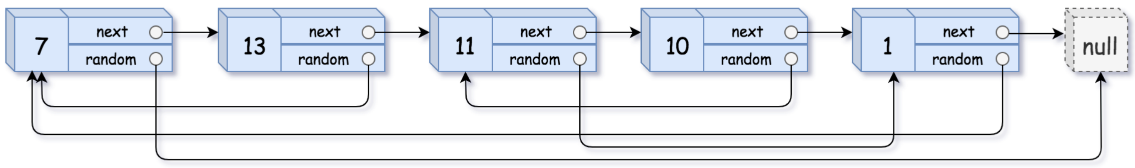
A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a **deep copy** of the list.

The Linked List is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

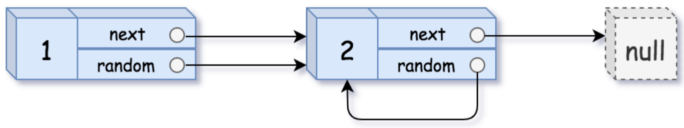
- `val`: an integer representing `Node.val`
- `random_index`: the index of the node (range from `0` to `n-1`) where random pointer points to, or `null` if it does not point to any node.

**Example 1:**



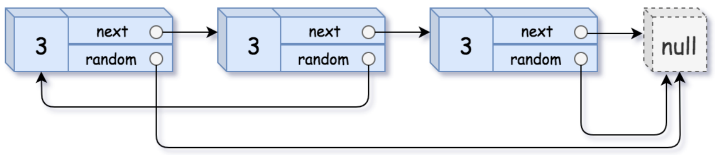
```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

**Example 2:**



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

**Example 3:**



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

**Example 4:**

```
Input: head = []
Output: []
Explanation: Given linked list is empty (null pointer), so return null.
```

```
class Solution {
    public Node copyRandomList(Node head) {
        // 总共三次遍历
        Node curr = head, copy = null, next = null;
        // 第一次遍历，复制结点，将复制的结点嵌入被复制的结点后
        while (curr != null) {
            next = curr.next;
            copy = new Node(curr.val);
            curr.next = copy;
            copy.next = next;
            curr = next;
        }
        curr = head;
        // 第二次遍历，复制random指针
        while (curr != null) {
            if (curr.random != null) {
                // 复制结点的random指向的就是被复制的结点的ramdom的复制结点
                curr.next.random = curr.random.next;
            }
            curr = curr.next.next;
        }
        Node dummy = new Node(0);
        Node copyIt = dummy;
        curr = head;
        // 第三次遍历，剥离出复制的链表
        while (curr != null) {
            copy = curr.next;
            copyIt.next = copy;
            copyIt = copy;
            curr.next = curr.next.next;
            curr = curr.next;
        }
        return dummy.next;
    }
}
```

## 150. Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

**Note:**

- Division between two integers should truncate toward zero.
- The given RPN expression is always valid. That means the expression would always evaluate to a result and there won't be any divide by zero operation.

**Example 1:**

```
Input: ["2", "1", "+", "3", "*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

**Example 2:**

```
Input: ["4", "13", "5", "/", "+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

```
class Solution {
    public int evalRPN(String[] tokens) {
        if (tokens == null || tokens.length == 0) {
            return 0;
        }
        // 操作数栈
        Deque<Integer> stack = new LinkedList<>();
        // 两个操作数
        int op1 = 0, op2 = 0;
        for (String s : tokens) {
            // 遇到操作符则从操作数栈中弹出两个数，进行计算后再压入栈
            // 注意：字符串是对象，比较它们的大小要用.equals()而不要用==
            if ("+".equals(s)) {
                stack.push(stack.pop() + stack.pop());
            } else if ("-".equals(s)) {
                op1 = stack.pop();
                op2 = stack.pop();
                stack.push(op2 - op1);
            } else if ("*".equals(s)) {
                stack.push(stack.pop() * stack.pop());
            } else if ("/".equals(s)) {
                op1 = stack.pop();
                op2 = stack.pop();
                stack.push(op2 / op1);
            } else {
                stack.push(Integer.parseInt(s));
            }
        }
        return stack.pop();
    }
}
```

## 34. Find First and Last Position of Element in Sorted Array

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

If the target is not found in the array, return `[-1, -1]`.

**Example 1:**

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

**Example 2:**

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        // 判断特殊情况
        if (nums == null || nums.length == 0) {
            return new int[]{-1, -1};
        }
        // 先二分查找，用左倾mid找出左侧边界
        int low = 0, high = nums.length - 1;
        while (low < high) {
            int mid = low + (high - low) / 2;
            // 想想只有两个数的情况， 8 8  target = 8
            if (target <= nums[mid]) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        // 如果找不到该元素，则说明不存在左边界和右边界
        if (nums[low] != target) {
            return new int[]{-1, -1};
        }
        int[] range = new int[2];
        // 赋值左边界
        range[0] = low;
        // low继续用上次二分的结果值
        // high重置
        high = nums.length - 1;
        // 右倾mid找出右边界
        while (low < high) {
            int mid = low + (high - low) / 2 + 1;
            // 还是想想只有两个数 8 8 target = 8的情况
            if (target >= nums[mid]) {
                low = mid;
            } else {
                high = mid - 1;
            }
        }
        range[1] = high;
        return range;
    }
}
```

## 79. Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

**Example:**

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

Given word = "ABCCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
```

**Constraints:**

- `board` and `word` consists only of lowercase and uppercase English letters.

```
class Solution {
    public boolean exist(char[][] board, String word) {
        if (board == null || board.length == 0 || board[0].length == 0) {
            return false;
        }
        char[] charArray = word.toCharArray();
        // 遍历每个字符，以遍历到的字符为首字母，进行探索
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (exist(board, charArray, i, j, 0)) {
                    return true;
                }
            }
        }
        return false;
    }

    private boolean exist(char[][] board, char[] word, int i, int j, int len) {
        // 当所有字符都匹配完之后，递归返回true
        if (len == word.length) {
            return true;
        }
        // 边界
        if (i < 0 || j < 0 || i >= board.length || j >= board[0].length) {
            return false;
        }
        // 探索到的字符与当前所需字符不匹配
        if (board[i][j] != word[len]) {
            return false;
        }
        // 探索周围的字符时，当前字符不能使用，所以用异或运算将当前字符置为无效
        board[i][j] ^= 128;
        // 往四个方向探索，只要有一个方向存在单词，则返回true
        boolean existed = exist(board, word, i + 1, j, len + 1)
            || exist(board, word, i - 1, j, len + 1)
            || exist(board, word, i, j + 1, len + 1)
            || exist(board, word, i, j - 1, len + 1);
        // 走到这里时，递归均已完成
        // 还原字符
        board[i][j] ^= 128;
        // 返回当前字符的遍历结果
        return existed;
    }
}
```

## 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 = 112
Output: 3
Explanation: 11 = 5 + 5 + 1
```

**Example 2:**

```
Input: coins = [2], amount = 3
Output: -1
```

**Note**:
You may assume that you have an infinite number of each kind of coin.

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        // 自底向上动态规划的思想
        // dp[i] 表示组合出数i所需的最少的硬币数
        int[] dp = new int[amount + 1];
        // 想求原问题，先求出子问题的解
        for (int i = 1; i <= amount; i++) {
            // min值
            int min = Integer.MAX_VALUE;
            // 遍历硬币面值
            for (int coin : coins) {
                // 如果当前数可以含有当前硬币
                // 且减去当前硬币面值后的数也可以由给出的硬币组成
                if (i - coin >= 0 && dp[i - coin] != -1) {
                    // 更新min
                    min = Math.min(dp[i - coin], min);
                }
            }
            // 如果min值未更新，说明当前数不能由硬币组合而成，赋值-1
            // 否则为之前的代价 + 本次选择的代价
            dp[i] = (min == Integer.MAX_VALUE) ? -1 : min + 1;
        }
        return dp[amount];
    }
}
```

# 19. Remove Nth Node From End of List

Given a linked list, remove the *n*-th node from the end of list and return its head.

**Example:**

```
Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
```

**Note:**

Given *n* will always be valid.

**Follow up:**

Could you do this in one pass?

```java
class Solution {
    public ListNode removeNthNodeFromEnd(ListNode head, int n) {
        // 快慢指针 + n gap
        // 哑结点
        ListNode dummy = new ListNode(0);
        // 刚开始两个指针都指向dummy
        ListNode slow = dummy, fast = dummy;
        // dummy下一个指向链表头结点
        dummy.next = head;
        // 让fast先走n + 1步， 使得slow和fast之间保持n个结点的距离
        for (int i = 0; i < n + 1; i++) {
            fast = fast.next;
        }
        // slow fast同时往后走， fast走到null的时候
        // slow刚好走到需要删除的结点的前面一个位置
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        // 返回dummy.next 因为head结点有可能会被删除 1->8 n = 2 / 1 n = 1
        return dummy.next;
    }
}
```

## 55. Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

**Example 1:**

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

**Example 2:**

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump
length is 0, which makes it impossible to reach the last index.
```

```
class Solution {
    public boolean canJump(int[] nums) {
        // last刚开始指向最后一个位置
        int last = nums.length - 1;
        // 从倒数第二个元素向前遍历
        for (int i = nums.length - 2; i >= 0; i--) {
            // 如果当前下标 + 最大步长 能够到达last的位置，就更新last的值
            if (i + nums[i] >= last) {
                last = i;
            }
        }
        // 如果last的值为0，则说明可以从开始的位置到达末尾点
        return last == 0;
    }
}
```

## 33. Search in Rotated Sorted Array

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., `[0,1,2,4,5,6,7]` might become `[4,5,6,7,0,1,2]`).

You are given a target value to search. If found in the array return its index, otherwise return `-1`.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

**Example 1:**

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

**Example 2:**

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

```
class Solution {
    public int search(int[] nums, int target) {
        // 第一次二分算法找到数组中的最小值
        int low = 0, high = nums.length - 1;
        while (low < high) {
            int mid = low + (high - low) / 2;
            // 如果mid右边有比mid小的数，则mid肯定不是最小值
            // 所以可以大胆舍弃mid，往右移
            if (nums[mid] > nums[high]) {
                low = mid + 1;
            } else {
                // 想想 0 1 的情况 mid->0 low->0 high->1
                high = mid;
            }
        }
        // 最小值的下标就是原数组旋转的len
        int rotateLen = low;
        // 重置low 和 high
        low = 0;
        high = nums.length - 1;
        // 普通的二分查找
        while (low <= high) {
            int mid = low + (high - low) / 2;
            // 通过rotateLen来还原原来的有序数组中的mid的下标
            int originalMid = (mid + rotateLen) % nums.length;
            if (target < nums[originalMid]) {
                high = mid - 1;
            } else if (target > nums[originalMid]) {
                low = mid + 1;
            } else {
                // 找到则返回
                return originalMid;
            }
        }
        return -1;
    }
}
```

## 54. Spiral Matrix

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

**Example 1:**

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

**Example 2:**

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

```
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> res = new LinkedList<>();
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return res;
        }
        int left = 0, right = matrix[0].length - 1;
        int top = 0, bottom = matrix.length - 1;
        while (true) {
            // 往右走
            for (int i = left; i <= right; i++) res.add(matrix[top][i]);
            top++;
            if (left > right || top > bottom) break;
            // 往下走
            for (int i = top; top <= bottom; i++) res.add(matrix[i][right]);
            right--;
            if (left > right || top > bottom) break;
            // 往左走
            for (int i = right; i >= left; i--) res.add(matrix[bottom][i]);
            bottom--;
            if (left > right || top > bottom) break;
            // 往上走
            for (int i = bottom; i >= top; i--) res.add(matrix[i][left]);
            left++;
            if (left > right || top > bottom) break;
        }
        return res;
    }
}
```

## 2. Add Two Numbers

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

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

**Example:**

```
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807.
```

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int sum = 0;
        ListNode dummy = new ListNode(0);
        ListNode curr = dummy;
        // 两个中有一个不为空，就继续处理
        while (l1 != null || l2 != null) {
            // 点睛之笔：将sum /= 10 得出低位数计算的结果的进位
            sum /= 10;
            // 若l1还有节点，累加节点的值，向后遍历
            if (l1 != null) {
                sum += l1.val;
                l1 = l1.next;
            }
            // 若l2还有节点，累加节点的值，向后遍历
            if (l2 != null) {
                sum += l2.val;
                l2 = l2.next;
            }
            // 创建新的节点，并让curr连接到新节点
            curr.next = new ListNode(sum % 10);
            // curr指向新的节点
            curr = curr.next;
        }
        // 边界情况：若处理完后最高位还有进位，则在更高的一位填充1
        if (sum / 10 == 1) {
            curr.next = new ListNode(1);
        }
        // 返回新形成的链表的头节点
        return dummy.next;
    }
}
```

## 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.

The cache is initialized with a **positive** capacity.

**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
```

```
class LRUCache {
    // 双向链表 + HashMap实现LRU
    private int capacity;
    // 带头尾结点
    private Node head = new Node(0, 0);
    private Node tail = new Node(0, 0);
```

```java
    // map: key -> node
    private Map<Integer, Node> map = new HashMap<>();

    public LRUCache(int capacity) {
        this.capacity = capacity;
        // 连接头尾结点
        head.next = tail;
        tail.prev = head;
    }

    public void put(int key, int value) {
        // 先生成一个node，因为之后insert和remove的操作单位都是node
        Node node = new Node(key, value);
        // 如果LRU中包含了该key,则先将其从map和双向链表中移除
        if (map.containsKey(key)) {
            remove(node);
        }
        // 若容量达到上限，则移除末尾结点
        if (map.size() == capacity) {
            remove(tail.prev);
        }
        // 在头部插入该结点
        insert(node);
    }

    public int get(int key) {
        // 如果不存在，直接返回-1
        if (!map.containsKey(key)) {
            return -1;
        }
        // 通过key获取node结点
        Node node = map.get(key);
        // 移除node
        remove(node);
        // 在首部插入node
        insert(node);
        // 返回node中的value
        return node.value;
    }

    private void insert(Node node) {
        // 在map中插入 key-> node
        map.put(node.key, node);
        // 插入到链表头
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    // 把node从map，和链表中移除
    private void remove(Node node) {
        map.remove(node.key);
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private class Node {
        int key;
        int value;
        Node prev;
        Node next;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

## 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.
```

```
class Solution {
    public int maxProduct(int[] nums) {
        // imax表示[0, i] 以i结尾的子数组的连续乘积的最大值
        // imin表示[0, i] 以i结尾的子数组的连续乘积的最小值
        int res = nums[0], imax = nums[0], imin = nums[0];
        for (int i = 1; i < nums.length; i++) {
            // 如果当前数为负数，与其相乘后最大值和最小值的会互相转换
            if (nums[i] < 0) {
                int temp = imax;
                imax = imin;
                temp = imax;
            }
            // 以i结尾的子数组的乘积最大值为前面的乘积 * 自身 或者自身 两者中的较大值
            imax = Math.max(imax * nums[i], nums[i]);
            // 以i结尾的子数组的乘积最小值为前面的乘积 * 自身 或者自身 两者中的较小值
            imin = Math.min(imin * nums[i], nums[i]);
            // 每个子数组的乘积的最大值都是全局最大值的一个候选
            res = Math.max(max, res);
        }
        return res;
    }
}
```

## 3. Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

**Example 1:**

```
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

**Example 2:**

```
Input: "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

**Example 3:**

```
Input: "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
             Note that the answer must be a substring, "pwke" is a subsequence
and not a substring.
```

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        // 判断特殊情况
        if (s == null || s.length() == 0) {
            return 0;
        }
        int maxLen = 0;
        // map中的key为字符串中的字符，value为下标
        Map<Character, Integer> map = new HashMap<>();
        // [left, right]中始终为只包含不重复字符的子串
        for (int left = 0, right = 0; right < s.length(); right++) {
            // 取出当前遍历到的字符
            char c = s.charAt(right);
            // 如果map中含有该字符，则说明是当前字符在子串中存在了
            if (map.containsKey(c) {
                // 需要调整left的位置，到重复的字符的后面一位
                // 以维持[left, right]这个子串都是不重复的字符
                left = Math.max(left, map.get(c) + 1);
            }
            // 放入映射关系
            map.put(c, right);
            // 更新maxLen
            maxLen = Math.max(maxLen, right - left + 1);
        }
        return maxLen;
    }
}
```

## 50. Pow(x, n)

Implement pow(*x*, *n*), which calculates *x* raised to the power *n* (xn).

**Example 1:**

```
Input: 2.00000, 10
Output: 1024.00000
```

**Example 2:**

```
Input: 2.10000, 3
Output: 9.26100
```

**Example 3:**

```
Input: 2.00000, -2
Output: 0.25000
Explanation: 2-2 = 1/22 = 1/4 = 0.25
```

```
class Solution {
    public int myPow(double x, int n) {
        // 递归边界
        if (n == 0) {
            return 1;
        }
        // 对 n < 0进行特殊处理
        if (n < 0) {
            return 1/x * myPow(1/x, -(n + 1));
        }
        // 若n>0, 分为n为奇偶两种情况
        return (n % 2 == 0) ? myPow(x*x, n/2) : x * myPow(x*x, n/2);
    }
}
```

## 127. Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list.

**Note:**

- Return 0 if there is no such transformation sequence.

- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

**Example 1:**

```
Input:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -
> "cog",
return its length 5.
```

**Example 2:**

```
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore no possible
transformation.
```

```
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String>
wordList) {
        // 若原字符串序列中不包含endWord说明不存在这样一个梯级序列
        if (!wordList.contains(endWord)) {
            return 0;
        }
        // 用set装原来list中的数据，避免时间复杂度过高
        Set<String> wordSet = new HashSet<>(wordList);
        Set<String> start = new HashSet<>();
        Set<String> end = new HashSet<>();
        // 开始扫描的word
        start.add(beginWord);
        // 结尾的word
        end.add(endWord);
        // 将beginWord和endWord从序列中删去
        wordSet.remove(beginWord);
        wordSet.remove(endWord);
        // 初始的序列长度为1，即只有beginWord
        int len = 1;
        // 遍历直到start集中的字符串为空
        while (!start.isEmpty()) {
            // 收集下一次要遍历的字符串
            Set<String> next = new HashSet<>();
            // 遍历start集中的所有字符串
            for (String word : start) {
                    // 将字符串串转化为字符数组
                    char[] chars = word.toCharArray();
                    // 遍历该字符数组
                    for (int i = 0; i < chars.length; i++) {
                        // 保存旧字符
                        char old = chars[i];
                        // 试着用26个字符去替代当前字符，以产生新字符串
                        for (char c = 'a'; c <= 'z'; c++) {
                            chars[i] = c;
                            String target = new String(chars);
                            // 若包含在endSet中，则说明已经找到序列，返回len + 1即可， + 1是
endSet中的该字符串也要算
                            if (end.contains(target)) {
                                return len + 1;
                            }
                            // 若在wordSet中包含该字符串，则将该字符串从wordSet转移到
nextSet中
                            if (wordSet.contains(target)) {
                                wordSet.remove(target);
                                next.add(target);
                            }
                        }
                        // 遍历完26个字母后还原字符，继续遍历下一个字符
                        chars[i] = old;
                    }
                }
                // 选取endset 和 nextset中较小的一个作为下一个start
                if (end.size() < next.size()) {
                    start = end;
                    end = next;
                } else {
                    start = next;
                }
                len++;
            }
            return 0;
        }
}
```

## 5. Longest Palindromic Substring

Given a string **s**, find the longest palindromic substring in **s**. You may assume that the maximum length of **s** is 1000.

**Example 1:**

```
Input: "babad"
Output: "bab"
Note: "aba" is also a valid answer.
```

**Example 2:**

```
Input: "cbbd"
Output: "bb"
```

```
class Solution {
    public String longestPalindrome(String s) {
        // 若只有0个或者1个字符，则直接返回
        if (s.length() < 2) {
            return s;
        }
        // 用数组，记录最长回文子串的起始与终止位置
        int[] maxStart = new int[1];
        int[] maxEnd = new int[1];
        // 遍历字符串，到倒数第二个字符为止
        for (int i = 0; i < s.length() - 1; i++) {
            // 判断是否存在个数为奇数个的回文子串
            extend(s, i, i, maxStart, maxEnd);
            // 判断是否存在个数为偶数个的回文子串
            extend(s, i, i + 1, maxStart, maxEnd);
        }
        // 返回最长回文子串
        return s.substring(maxStart[0], maxEnd[0] + 1);
    }

    private void extend(String s, int left, int right, int[] maxStart, int[]
maxEnd) {
        // left向左 right向右 找到回文子串
        while (left >= 0 && right < s.length() && s.charAt(left) ==
s.charAt(right)) {
            left--;
            right++;
        }
        // 让left和right回到合法的回文子串的位置
        left++;
        right--;
        // 更新最长回文子串的位置
        if (right - left > maxEnd[0] - maxStart[0]) {
            maxStart[0] = left;
            maxEnd[0] = right;
        }
    }
}
```

## 179. Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

**Example 1:**

```
Input: [10,2]
Output: "210"
```

**Example 2:**

```
Input: [3,30,34,5,9]
Output: "9534330"
```

**Note:** The result may be very large, so you need to return a string instead of an integer.

```
class Solution {
    public String largestNumber(int[] nums) {
        // 判断特例
        if (nums == null || nums.length == 0) {
            return "";
        }
        // 将数字数字转换为字符串数组
        String[] strs = new String[nums.length];
        for (int i = 0; i < nums.length; i++) {
            strs[i] = String.valueOf(nums[i]);
        }
        // 利用一定的规则给字符串数组中的元素排序
        // int Compare方法 返回< 0表示s1 < s2，所以s1 要放在s2前面
        Arrays.sort(strs, (s1, s2) -> (s2 + s1).compareTo(s1 + s2));
        // 若输入的元素全部是0 0 0 0 0（只要有一个不是0，第一个字符都不可能为0），则直接返回
0
        if (strs[0].charAt(0) == '0') {
            return "0";
        }
        // 再将这些字符串拼接起来
        StringBuilder res = new StringBuilder();
        for (String str : strs) {
            res.append(str);
        }
        return res.toString();
    }
}
```

# 130. Surrounded Regions

Given a 2D board containing `'X'` and `'O'` (**the letter O**), capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.

**Example:**

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

**Explanation:**

Surrounded regions shouldn’t be on the border, which means that any `'O'` on the border of the board are not flipped to `'X'`. Any `'O'` that is not on the border and it is not connected to an `'O'` on the border will be flipped to `'X'`. Two cells are connected if they are adjacent cells connected horizontally or vertically.

```
class Solution {
    public void solve(char[][] board) {
        // 判断特例
        if (board == null || board.length == 0 || board[0].length == 0) {
            return;
        }
        int m = board.length;
        int n = board[0].length;
        // 对左右两列进行dfs，把与边相邻的 O -> *
        for (int i = 0; i < m; i++) {
            dfs(board, i, 0);
            dfs(board, i, n - 1);
        }
        // 对上下两行进行dfs
        for (int j = 1; j < n - 1; j++) {
            dfs(board, 0, j);
            dfs(board, m - 1, j);
        }
        // 遍历整个矩阵，遇到O(说明是里面的 不靠近边界的O) -> X
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 'O') {
                    board[i][j] = 'X';
                }
                // 遇到* 还原为O
                if (board[i][j] == '*') {
                    board[i][j] = 'O';
                }
            }
        }
    }

    private void dfs(char[][] board, int i, int j) {
        // 边界条件的限制
        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length) {
            return;
        }
        // 如果遇到X 或者是已经标记过的点
        if (board[i][j] == 'X' || board[i][j] == '*') {
            return;
        }
        // 将边界的O -> *
        board[i][j] = '*';
        // dfs遍历其余格
```

```
            dfs(board, i + 1, j);
            dfs(board, i - 1, j);
            dfs(board, i, j + 1);
            dfs(board, i, j - 1);
        }
    }
```

## 98. Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

**Example 1:**

```
    2
   / \
  1   3

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

**Example 2:**

```
    5
   / \
  1   4
     / \
    3   6

Input: [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

```
class Solution {
    // recursively
    public boolean isValidBST(TreeNode root) {
        return helper(root, null, null);
    }

    private boolean helper(TreeNode root, TreeNode min, TreeNode max) {
        // 空树为二叉搜索树
        if (root == null) {
            return true;
        }
        // 若当前的结点的值 >= 其右边的父结点的值 或者 <= 左边的父结点的值 则不是一个BST
        if ((max != null && root.val >= max.val) || (min != null && root.val <=
min.val)) {
            return false;
        }
        // 利用递归中的参数做赋值操作
        // root = root.left;
        // min = min;
        // max = root;
        return helper(root.left, min, root) && helper(root.right, root, max);
    }

    // iteratively
    // BST的中序遍历序列应该是有序的
    public boolean isValidBST(TreeNode root) {
        if (root == null) {
            return true;
        }
        TreeNode curr = root;
        TreeNode prev = null;
        Deque<TreeNode> stack = new LinkedList<>();
        while (curr != null || !stack.isEmpty()) {
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }
            curr = stack.pop();
            // 若前驱结点存在且当前结点的值 <= 前驱结点的值，则不是一颗BST
            if (prev != null && curr.val <= prev.val) {
                return false;
            }
            prev = curr;
            curr = curr.right;
        }
        return true;
    }
}
```

# 15. 3Sum

Given an array `nums` of *n* integers, are there elements *a*, *b*, *c* in `nums` such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

**Note:**

The solution set must not contain duplicate triplets.

**Example:**

```
Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        // 先将数组中的元素排序
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 2; i++) {
            // 剪枝：若当前元素>0，则后面的元素都大于0，其和不可能为0
            if (nums[i] > 0) {
                break;
            }
            // 为了避免重复：保证重复的元素中只有第一个会进入处理逻辑
            if (i == 0 || nums[i] != nums[i - 1]) {
                // 将3个数相加的和 降维为2个数相加（神来之笔）
                int sum = 0 - nums[i];
                // 用两个下标查找匹配的数
                int lo = i + 1, hi = nums.length - 1;
                while (lo < hi) {
                    // 因为数组中的元素已经有序
                    // 若小于目标，则lo右移，使得和增大
                    if (nums[lo] + nums[hi] < sum) {
                        lo++;
                        // 若大于目标，则hi左移，使得和减少
                    } else if (nums[lo] + nums[hi] > sum) {
                        hi--;
                    } else {
                        // 当遇到匹配的数时，添加入res中
                        res.add(Arrays.asList(nums[i], nums[lo], nums[hi]));
                        // 同样的也要保证第二个和第三个元素中
                        // 若有重复的 只取首次出现的一个
                        while (lo < hi && nums[lo] == nums[lo + 1]) lo++;
                        while (lo < hi && nums[hi] == nums[hi - 1]) hi--;
                        // 同时移动lo hi 寻找下一个匹配的
                        lo++;
                        hi--;
                    }
                }
            }
        }
        return res;
    }
}
```

# 91. Decode Ways

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given a **non-empty** string containing only digits, determine the total number of ways to decode it.

**Example 1:**

```
Input: "12"
Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
```

**Example 2:**

```
Input: "226"
Output: 3
Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
```

```java
class Solution {
    public int numDecodings(String s) {
        // 判断特例
        if (s == null || s.length() == 0) {
            return 0;
        }
        int len = s.length();
        // 状态：dp[i] 表示 [i, len - 1]这个子串可以被解码的方式有多少种
        int[] dp = new int[len + 1];
        // 边界
        dp[len] = 1;
        dp[len - 1] = s.charAt(len - 1) != '0' ? 1 : 0;
        // 因为至少要判断2位数，所以从倒数第二位开始往前遍历
        for (int i = len - 2; i >= 0; i--) {
            // 如果是0， 则它自身不能被解码，它与后面的数组合在一起也不可能被解码
            // 所以直接跳过
            if (s.charAt(i) == '0') {
                continue;
            }
            // 状态转移方程：如果这个数和之后的数组合起来可以被编码，其值等于后面两个数的编码
方式之和
            if (Integer.parseInt(s.substring(i, i + 2)) <= 26) {
                dp[i] = dp[i + 1] + dp[i + 2];
            } else {
                // 否则就直接继承
                dp[i] = dp[i + 1];
            }
        }
        // 返回[0, len - 1]这个子串的被编码的方式
        return dp[0];
    }
}
```

## 166. Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

If multiple answers are possible, just return any of them.

**Example 1:**

```
Input: numerator = 1, denominator = 2
Output: "0.5"
```

**Example 2:**

```
Input: numerator = 2, denominator = 1
Output: "2"
```

**Example 3:**

```
Input: numerator = 2, denominator = 3
Output: "0.(6)"
```

```
class Solution {
    // 就用 -2 3 去模拟整个算法过程
    public String fractionToDecimal(int numerator, int denominator) {
        // 若被除数为0，直接返回0对应的字符串
        if (numerator == 0) {
            return "0";
        }
        StringBuilder sb = new StringBuilder();
        // 取出符号，位运算符也可以用于boolean变量，true ^ false -> true
        String sign = (numerator > 0) ^ (denominator) ? "-" : "";
        res.append(sign);
        // 转换为long， 避免计算中的溢出
        long num = Math.abs((long)numerator);
        long den = Math.abs((long)denominator);
        // 取出整数部分
        res.append(num / den);
        // 余数
        num %= den;
        if (num == 0) {
            res.toString();
        }
        res.append(".");
        Map<Long, Integer> map = new HashMap<>();
        // 记下该余数和其对应的下标（即之后的插入位置）
        map.put(num, res.length());
        while (num != 0) {
            // 2->20
            num *= 10;
            // 20 / 3
            res.append(num / den);
            // num -> 2
            num %= den;
            // 若该余数之前已经出现，则在Index处添加左括号 在末尾添加右括号
            if (map.containsKey(num)) {
                int index = map.get(num);
                res.insert(index, "(");
                res.append(")");
                break;
            } else {
                map.put(num, res.length());
            }
        }
        return res.toString();
    }
}
```

## 29. Divide Two Integers

Given two integers `dividend` and `divisor`, divide two integers without using multiplication, division and mod operator.

Return the quotient after dividing `dividend` by `divisor`.

The integer division should truncate toward zero, which means losing its fractional part. For example, `truncate(8.345) = 8` and `truncate(-2.7335) = -2`.

**Example 1:**

```
Input: dividend = 10, divisor = 3
Output: 3
Explanation: 10/3 = truncate(3.33333..) = 3.
```

**Example 2:**

```
Input: dividend = 7, divisor = -3
Output: -2
Explanation: 7/-3 = truncate(-2.33333..) = -2.
```

**Note:**

- Both dividend and divisor will be 32-bit signed integers.
- The divisor will never be 0.
- Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231,  231 − 1]. For the purpose of this problem, assume that your function **returns 231 − 1 when the division result overflows**.

```
class Solution {
    public int divide(int dividend, int divisor) {
        // 整数除法中唯一的溢出情况就是 -2^31 / -1 => 2^31（溢出）
        if (divident == 1 << 31 && divisor == -1) {
            return (1 << 31) - 1;
        }
        // 取绝对值进行计算
        int a = Math.abs(dividend), b = Math.abs(divisor);
        int res = 0, x = 0;
        while (a - b >= 0) {
            for (x = 0; a - (b << x << 1) >= 0; x++);

            res += 1 << x;
            a -= b << x;
        }
        // 判断符号
        return (dividend > 0) == (divisor > 0) ? res : -res;
    }
}
```

# 8. String to Integer (atoi)

Implement `atoi` which converts a string to an integer.

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned.

**Note:**

- Only the space character `' '` is considered as whitespace character.
- Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231, 231 − 1]. If the numerical value is out of the range of representable values, INT_MAX (231 − 1) or INT_MIN (−231) is returned.

**Example 1:**

```
Input: "42"
Output: 42
```

**Example 2:**

```
Input: "   -42"
Output: -42
Explanation: The first non-whitespace character is '-', which is the minus sign.
             Then take as many numerical digits as possible, which gets 42.
```

**Example 3:**

```
Input: "4193 with words"
Output: 4193
Explanation: Conversion stops at digit '3' as the next character is not a
numerical digit.
```

**Example 4:**

```
Input: "words and 987"
Output: 0
Explanation: The first non-whitespace character is 'w', which is not a numerical

             digit or a +/- sign. Therefore no valid conversion could be
performed.
```

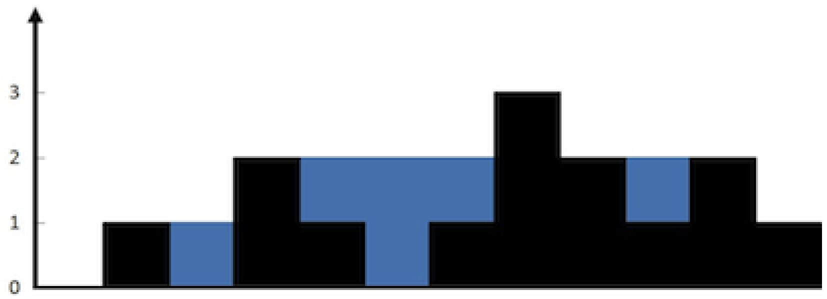
**Example 5:**

```
Input: "-91283472332"
Output: -2147483648
Explanation: The number "-91283472332" is out of the range of a 32-bit signed
integer.
             Thefore INT_MIN (−231) is returned.
```

```java
class Solution {
    public int myAtoi(String str) {
        // 判断特例
        if (str == null || str.length() == 0) {
            return 0;
        }
        // 下标， 符号， 结果
        int i = 0, sign = 1, res = 0;
        // 跳过前面的空格
        while (i < str.length() && str.charAt(i) == ' ') {
            i++;
        }
        // 如果字符串中只有空格，直接返回0
        if (i >= str.length()) {
            return 0;
        }
        // 判断并跳过符号位
        if (str.charAt(i) == '+') {
            i++;
        } else if (str.charAt(i) == '-') {
            sign = -1;
            i++;
        }
        // 计算有效位数开始的计数 如可能会有 0000000000123影响计数
        int validCount = 0;
        // 遍历直到i遍历完字符串，或者该字符不是数字
        while (i < str.length() && Character.isDigit(str.charAt(i))) {
            // 取出该数字
            int num = str.charAt(i) - '0';
            // 记录下计算之前的数值
            int temp = res;
            // 加上该位数
            res = res * 10 + num;
            // 累加有效位数
            if (res != 0) {
                validCount++;
            }
            // 如果有效位数>=10，则说明有可能发生溢出
            // res < 0 或者 (res - num) / 10 != 原来的数都说明发生了溢出现象
            if (validCount > 9 && (res < 0 || (res - num) / 10 != temp)) {
                return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
            }
            // 移动下标
            i++;
        }
        // 赋值符号
        return sign * res;
    }
}
```

## 42. Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!

**Example:**

```
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
```

```
class Solution {
    /*
    Here is my idea: instead of calculating area by height*width, we can think it
in a cumulative way. In other words, sum water amount of each bin(width=1).
Search from left to right and maintain a max height of left and right separately,
which is like a one-side wall of partial container. Fix the higher one and flow
water from the lower part. For example, if current height of left is lower, we
fill water in the left bin. Until left meets right, we filled the whole
container.
    */
    public int trap(int[] height) {
        // 左右两个指针，向中间走
        int i = 0, j = height.length - 1;
        // 累积计算积水的体积，  当前的左边墙的最大高度，当前右边墙的最大高度
        int sum = 0, maxLeft = 0, maxRight = 0;
        // 循环直到两个指针相遇
        while (i < j) {
            // 选择当前高度较小的一方进行计算（因为另一方比它高，可以容纳下水）
            if (height[i] <= height[j]) {
                // 如果当前高度>=最大高度，更新最大高度
                if (height[i] >= maxLeft) {
                    maxLeft = height[i];
                } else {
                    // 如果当前高度小于当前边的最大高度，则肯定有积水
                    // 而且第一个判断条件保证了积水一定可以容得下
                    sum += maxLeft - height[i];
                }
                // 移动左指针
                i++;
            } else {
                // 右指针处理思路同上
                if (height[j] >= maxRight) {
                    maxRight = height[j];
                } else {
                    sum += maxRight - height[j];
                }
                j--;
            }
        }
        // 返回总共的积水
        return sum;
    }
}
```

## 297. Serialize and Deserialize Binary Tree

Hard

3266159Add to ListShare

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.

```java
class Codec {
    // 序列化后的分隔符
    private static String separator = ",";
    // null结点代表的字符串
    private static String nullNode = "X";

   public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
      // 构造字符串
        buildString(root, sb);
        return sb.toString();
    }

    public TreeNode deserialize(String data) {
        // 将字符串根据分隔符，转化为字符串数组
        List<String> strs = Arrays.asList(data.split(separator));
        Queue<String> queue = new LinkedList<>();
        // 将list中的字符串放到queue中
        queue.addAll(strs);
        // 先序遍历递归构造树
        TreeNode root = buildTree(queue);
        return root;
    }

    private TreeNode buildTree(Queue<String> queue) {
        // 取出队列头
        String value = queue.poll();
        // 若为代表nullNode的X 则返回null
        if (nullNode.equals(value)) {
            return null;
        }
        // 若为数字的字符串表示，则转化为数字，并添加为树中结点的val
        TreeNode root = new TreeNode(Integer.parseInt(value));
        // 递归构造左子树
        root.left = buildTree(queue);
        // 递归构造右子树
        root.right = buildTree(queue);
        // 返回根结点
        return root;
    }

    private void buildString(TreeNode root, StringBuilder sb) {
        // 先序遍历，递归边界：遇到null则填充X，并返回
        if (root == null) {
            sb.append(nullNode).append(separator);
            return;
        }
        // 结点不是null，则添加其val到字符串中
        sb.append(root.val).append(separator);
        // 递归遍历左子树
        buildString(root.left, sb);
        // 递归遍历右子树
        buildString(root.right, sb);
    }
}
```

## 128. Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

Your algorithm should run in O(*n*) complexity.

**Example:**

```
Input: [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore
its length is 4.
```

```
class Solution {

    /*
    We will use HashMap. The key thing is to keep track of the sequence length
and store that in the boundary points of the sequence. For example, as a result,
for sequence {1, 2, 3, 4, 5}, map.get(1) and map.get(5) should both return 5.

Whenever a new element n is inserted into the map, do two things:

See if n - 1 and n + 1 exist in the map, and if so, it means there is an existing
sequence next to n. Variables left and right will be the length of those two
sequences, while 0 means there is no sequence and n will be the boundary point
later. Store (left + right + 1) as the associated value to key n into the map.
Use left and right to locate the other end of the sequences to the left and right
of n respectively, and replace the value with the new length.
    */
    public int longestConsecutive(int[] nums) {
        // 判断特例
        if (nums == null || nums.length == 0) {
            return 0;
        }
        // 最长连续子序列的长度
        int maxLen = 0;
        // 里面存的是数组中不重复的数值 -> 其所在的子序列的长度（只有子序列左右两端会更新）
        Map<Integer, Integer> map = new HashMap<>();
        for (int n : nums) {
            // 对于重复数字，因为已经处理过了，所以不能进行重复的计算，直接跳过
            if (map.containsKey(n)) {
                continue;
            }
            // 尝试获得其左边的子序列的长度 没有则为0
            int left = map.getOrDefault(n - 1, 0);
            // 尝试获得该数右边的子序列的长度，没有则为0
            int right = map.getOrDefault(n + 1, 0);
            // 该数所在的子序列的长度 = 左边的子序列的长度 + 右边的子序列的长度 + 自己
            int len = left + right + 1;
            // 更新最大长度
            maxLen = Math.max(maxLen, len);
            // 设置该数对应的子序列的长度
            map.put(n, len);
            // 设置该子序列两端的位置的子序列长度（保持同步）
            map.put(n - left, len);
            map.put(n + right, len);
        }
        return maxLen;
    }
}
```

# 295. Find Median from Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

For example,

```
[2,3,4]`, the median is `3
[2,3]`, the median is `(2 + 3) / 2 = 2.5
```

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

**Example:**

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

**Follow up:**

1. If all integer numbers from the stream are between 0 and 100, how would you optimize it?
2. If 99% of all integer numbers from the stream are between 0 and 100, how would you optimize it?

```java
class MedianFinder {
    // 左边用一个逻辑上的最大堆（实际上是最小堆）（里面存储的是比较小的那一部分数的负数形式）
    // 所以堆顶元素就是较小的那一半的最大元素（-1 -2）->（1 2）
    private Queue<Long> small = new PriorityQueue<>();
    // 右边用一个最小堆存储较大的那一半的元素，堆顶元素就是最小元素 (3 4)
    private Queue<Long> large = new PriorityQueue<>();
    // 所以通过访问两个堆的堆顶元素就可以获得中间数或者中间数的两边

    // 使用Long的原因：
    // 1. 用int的话-2^31 取反的时候会溢出
    // 2. 用int的话 在进行 large.peek() - small.peek() 的时候也有可能发生溢出

    // 插入操作O(logn)
    public void addNum(int num) {
        // 放入large
        large.offer((long)num);
        // 取负数 倒腾到small
        small.offer(-large.poll());
        // 如果small的size比large大了 再倒腾回来
        if (small.size() > large.size()) {
            large.offer(-small.poll());
        }
    }
    // 获取中间元素操作O(1)
    public double findMedian() {
        // 如果large中的元素多一个 那就说明当前的元素个数为奇数
        // 返回较大的那一半的堆顶元素（即中间元素）
        if (large.size() > small.size()) {
            return large.peek();
        }
        // 如果两个堆中的元素相等 那就返回堆顶元素之差的一半（逻辑上是之和）
        return (large.peek() - small.peek()) / 2.0;
    }
}
```