

LeetCode每日一题高频面试算法题目

day01 （队列实现栈）

使用队列实现栈的下列操作：

- push(x) -- 元素 x 入栈
- pop() -- 移除栈顶元素
- top() -- 获取栈顶元素
- empty() -- 返回栈是否为空

注意：

- 你只能使用队列的基本操作-- 也就是 push to back , peek/pop from front , size , 和 is empty 这些操作是合法的。
- 你所使用的语言也许不支持队列。 你可以使用 list 或者 deque（双端队列）来模拟一个队列，只要是标准的队列操作即可。
- 你可以假设所有操作都是有效的（例如，对一个空的栈不会调用 pop 或者 top 操作）。

```
class MyStack {
    private Queue<Integer> q1; //输入
    private Queue<Integer> q2; //输出
    private int top;
    /** Initialize your data structure here. */
    public MyStack() {
        q1 = new LinkedList<Integer>();
        q2 = new LinkedList<Integer>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        q1.offer(x); //q1接收数据，将q2中所有数据复制到q1，保证先来的数据在后面，新来的数据在前面
        while(!q2.isEmpty()){
            q1.offer(q2.poll()); //poll返回链表头并删除    offer插入链表尾部
        }
        Queue temp = q1;
        q1 = q2;
        q2 = temp;
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        return q2.poll();
    }

    /** Get the top element. */
    public int top() {
        return q2.peek();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
```

```

        return q2.isEmpty();
    }
}

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * boolean param_4 = obj.empty();
 */

```

day02 (反转一个单链表)

反转一个单链表。

示例:

```

输入：1->2->3->4->5->NULL
输出：5->4->3->2->1->NULL

```

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题？

非递归

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next==null)
            return head;
        ListNode curNode = head.next;
        ListNode preNode = head;
        preNode.next = null;
        ListNode curNext;
        while(curNode!=null){
            curNext = curNode.next;
            curNode.next = preNode;
            preNode = curNode;
            curNode = curNext;
        }
        return preNode;
    }
}

```

时间复杂度: $O(n)$, 假设 nn 是列表的长度, 时间复杂度是 $O(n)$ 。

空间复杂度: $O(1)$ 。

递归版本稍微复杂一些，其关键在于反向工作。假设列表的其余部分已经被反转，现在我该如何反转它前面的部分？

假设列表为：

$$n_1 \rightarrow \dots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \rightarrow \dots \rightarrow n_m \rightarrow \emptyset$$

若从节点 n_{k+1} 到 n_m 已经被反转，而我们正处于 n_k 。

$$n_1 \rightarrow \dots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \leftarrow \dots \leftarrow n_m$$

我们希望 n_{k+1} 的下一个节点指向 n_k 。

所以， $n_k.next.next = n_k$ 。

要小心的是 n_1 的下一个必须指向 \emptyset 。如果你忽略了这一点，你的链表中可能会产生循环。如果使用大小为 2 的链表测试代码，则可能会捕获此错误。

//1->2->3->4->5:递归执行完向下走的时候，第一次的p指向5，head指向4,head.next是5，当执行head.next.next=head时，p.next指向4，当执行head.next=null时，断开head的4到5的节点完成一次反转，以此类推

```
public ListNode reverseList(ListNode head){
    if(head==null || head.next==null){
        return head;
    }
    ListNode p = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return p;
}
```

时间复杂度： $O(n)$ ，假设 nn 是列表的长度，那么时间复杂度为 $O(n)$ 。

空间复杂度： $O(n)$ ，由于使用递归，将会使用隐式栈空间。递归深度可能会达到 n 层。

day03（合并两个排序数组保证有序）

给定两个排序后的数组 A 和 B，其中 A 的末端有足够的缓冲空间容纳 B。编写一个方法，将 B 合并入 A 并排序。

初始化 A 和 B 的元素数量分别为 m 和 n 。

示例：

输入：

A = [1,2,3,0,0,0], m = 3

B = [2,5,6], n = 3

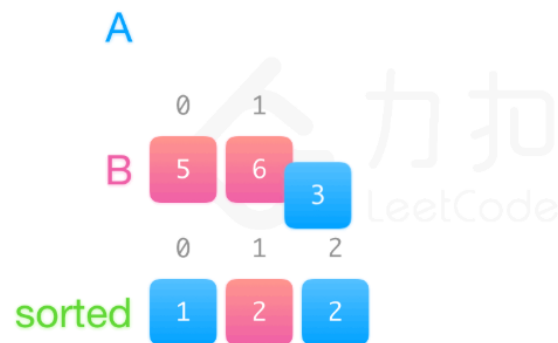
输出：[1,2,2,3,5,6]

解析：

方法 2: 双指针

算法

方法 1 没有利用数组 A 与 B 已经被排序的性质。为了利用这一性质，我们可以使用双指针方法。这一方法将两个数组看作队列，每次从两个数组头部取出比较小的数字放到结果中。如下面的动画所示：



```
class Solution {
    public void merge(int[] A, int m, int[] B, int n) {
        int pa = 0;
        int pb = 0;
        int[] sorted = new int[m+n];
        int curVal = 0;
        while(pa < m || pb < n){
            if(pa == m){ //注意判断其中一个是否到达尾部
                curVal = B[pb++];
            } else if(pb == n){
                curVal = A[pa++];
            } else if(A[pa] < B[pb]){
                curVal = A[pa++];
            } else{
                curVal = B[pb++];
            }
            sorted[pa+pb-1] = curVal;
        }
        for(int i=0; i < m+n; i++){
            A[i] = sorted[i];
        }
    }
}
```

day04 (腐烂的橘子)

994. 腐烂的橘子

难度 简单

146



在给定的网格中，每个单元格可以有以下三个值之一：

- 值 0 代表空单元格；
- 值 1 代表新鲜橘子；
- 值 2 代表腐烂的橘子。

每分钟，任何与腐烂的橘子（在 4 个正方向上）相邻的新鲜橘子都会腐烂。

返回直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 -1。

示例 1：



输入: `[[2,1,1],[1,1,0],[0,1,1]]`

输出: 4

- 1、先将所有腐烂橘子放入Queue (LinkedList) 中，建立Map `key=r*C+c` `value=此时此刻腐烂橘子所经历的时间`
- 2、当queue不为空 循环遍历，queue remove得到腐烂橘子队列中的位置，分析该腐烂橘子上下左右使其腐烂，并把腐烂橘子 (`key=r*C+c`, `value=上层腐烂橘子对应时间+1`)
- 3、遍历网格，如果有位置为1，说明有橘子未腐烂，return -1，否则返回map中的最大value

```
class Solution {
    //对行和列进行移动，上，左，下，右
    int[] dr = new int[]{-1,0,1,0};
    int[] dc = new int[]{0,-1,0,1};
    public int orangesRotting(int[][] grid) {
        int R = grid.length;
        int C = grid[0].length;

        Queue<Integer> queue = new LinkedList();
        Map<Integer,Integer> depth = new HashMap<>();
        //先遍历寻找该开始就腐烂的橘子
        for(int r=0;r<R;r++){
            for(int c=0;c<C;c++){
                if(grid[r][c]==2){
                    int code = r*C+c; //将表格中腐烂橘子的二维坐标转化为一个数字编码
                    queue.add(code);
                    depth.put(code,0); //key为二维坐标对应的数字编码，value为该编码对
应的橘子腐烂用时
                }
            }
        }
    }
}
```

```

int ans = 0;
while(!queue.isEmpty()){
    int code = queue.remove();
    int r = code/C;
    int c = code%C;
    for(int k=0;k<4;k++){ //将该腐烂橘子的上下左右依次腐烂
        int nr = r + dr[k];
        int nc = c + dc[k];
        if(nr>=0 && nr<R && nc>=0 && nc<C && grid[nr][nc]==1){
            grid[nr][nc] = 2;
            int ncode = nr*C+nc;
            queue.add(ncode);
            depth.put(ncode,depth.get(code)+1); //对腐烂橘子的时刻进行重新设定
            ans = depth.get(ncode);
        }
    }
}

for(int[] r:grid){
    for(int c :r){
        if(c==1){
            return -1;
        }
    }
}
return ans;
}
}

```

时间复杂度: $O(nm)$

即进行一次广度优先搜索的时间, 其中 $n=grid.length$, $m=grid[0].length$

空间复杂度: $O(nm)$

需要额外的 `dis` 数组记录每个新鲜橘子被腐烂的最短时间, 大小为 $O(nm)$, 且广度优先搜索中队列里存放的状态最多不会超过 nm 个, 最多需要 $O(nm)$ 的空间, 所以最后的空间复杂度为 $O(nm)$ 。

day05 (分糖果II)

1103. 分糖果 II

难度 简单  45     

排排坐，分糖果。

我们买了一些糖果 `candies`，打算把它们分给排好队的 $n = \text{num_people}$ 个小朋友。

给第一个小朋友 1 颗糖果，第二个小朋友 2 颗，依此类推，直到给最后一个小朋友 n 颗糖果。

然后，我们再回到队伍的起点，给第一个小朋友 $n + 1$ 颗糖果，第二个小朋友 $n + 2$ 颗，依此类推，直到给最后一个小朋友 $2 * n$ 颗糖果。

重复上述过程（每次都比上一次多给出一颗糖果，当到达队伍终点后再次从队伍起点开始），直到我们分完所有的糖果。注意，就算我们手中的剩下糖果数不够（不比前一次发出的糖果多），这些糖果也会全部发给当前的小朋友。

返回一个长度为 `num_people`、元素之和为 `candies` 的数组，以表示糖果的最终分发情况（即 `ans[i]` 表示第 i 个小朋友分到的糖果数）。

示例 1：

输入：candies = 7, num_people = 4

输出：[1,2,3,1]

解释：

第一次，`ans[0] += 1`，数组变为 [1,0,0,0]。

第二次，`ans[1] += 2`，数组变为 [1,2,0,0]。

第三次，`ans[2] += 3`，数组变为 [1,2,3,0]。

第四次，`ans[3] += 1`（因为此时只剩下 1 颗糖果），最终数组变为 [1,2,3,1]。

```
class Solution {
    public int[] distributeCandies(int candies, int num_people) {
        int[] ans = new int[num_people];
        int i = 0;
        while(candies!=0){
            ans[i%num_people] += Math.min(candies,i+1); //这里i%num_people用来多次循环数组
            candies -= Math.min(candies,i+1);
            i +=1;
        }
        return ans;
    }
}
```

day06（和为s的连续正数序列）

面试题57 - II. 和为s的连续正数序列

难度 简单

👍 58

❤

📄

🔍

🔔

💬

输入一个正整数 `target`，输出所有和为 `target` 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例 1:

输入: `target = 9`
输出: `[[2,3,4],[4,5]]`

示例 2:

输入: `target = 15`
输出: `[[1,2,3,4,5],[4,5,6],[7,8]]`

限制:

- $1 \leq \text{target} \leq 10^5$

Idea: (滑动数组)

- 1、因为 $\text{target} \geq 1$ 所以滑动数组左右边界初始化为1
- 2、规定滑动数组尺寸 $[i,j)$ 左闭右开
- 3、循环次数为 $i \leq \text{target}/2$ 因为数组连续，当滑动数组最左边 $\geq \text{target}/2$ 时，数组无论如何也不会出现和为`target`组合

```
class Solution {
    public int[][] findContinuousSequence(int target) {
        //滑动窗口解决问题
        int i = 1; //窗口最左端
        int j = 1; //窗口最右端 [i,j)
        List<int[]> ans = new ArrayList<>();
        int sum = 0;
        while(i <= target/2){
            if(sum < target){
                sum += j;
                j++;
            }else if(sum > target){
                sum -= i;
                i++;
            }else{
                int[] temp = new int[j-i];
                for(int k=i;k<j;k++){
                    temp[k-i] = k;
                }
                ans.add(temp);
                sum -= i;
            }
        }
    }
}
```



```

        i++;
    }
}
return ans.toArray(new int[ans.size()][]);
}
}

```

day07 (队列的最大值)

面试题59 - II. 队列的最大值

难度 中等  68     

请定义一个队列并实现函数 `max_value` 得到队列里的最大值，要求函数 `max_value`、`push_back` 和 `pop_front` 的均摊时间复杂度都是 $O(1)$ 。

若队列为空，`pop_front` 和 `max_value` 需要返回 -1

示例 1：

输入：
 ["MaxQueue", "push_back", "push_back", "max_value", "pop_front", "max_value"]
 [[], [1], [2], [], [], []]
 输出：[null, null, null, 2, 1, 2]

示例 2：

输入：
 ["MaxQueue", "pop_front", "max_value"]
 [[], [], []]
 输出：[null, -1, -1]

限制：

- $1 \leq \text{push_back, pop_front, max_value 的总操作数} \leq 10000$
- $1 \leq \text{value} \leq 10^5$

idea:

- 1、初始化一个队列存放每次push_back的数据
- 2、初始化一个双端队列，要求该队列front-->back为递减，保证front始终为最大值

```

class MaxQueue {
    private Queue<Integer> queue = new LinkedList<>();
    private Deque<Integer> deque = new ArrayDeque<>();
    public MaxQueue() {

    }

    public int max_value() {
        return deque.isEmpty() ? -1:deque.peekFirst();
    }
}

```

```

    }

    public void push_back(int value) {
        queue.offer(value);
        //deque成为递减队列
        while(!deque.isEmpty() && deque.peekLast() < value){
            deque.pollLast();
        }
        deque.offerLast(value);
    }

    public int pop_front() {
        Integer result = queue.poll(); //queue存放入队列的各个数据
        if(!deque.isEmpty() && deque.peekFirst().equals(result)){
            deque.pollFirst();
        }
        return result == null ? -1 : result;
    }
}

/**
 * Your MaxQueue object will be instantiated and called as such:
 * MaxQueue obj = new MaxQueue();
 * int param_1 = obj.max_value();
 * obj.push_back(value);
 * int param_3 = obj.pop_front();
 */

```

day08 (零钱兑换)

322. 零钱兑换

难度 中等  463     

给定不同面额的硬币 `coins` 和一个总金额 `amount`。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

示例 1:

输入: `coins = [1, 2, 5]`, `amount = 11`
 输出: 3
 解释: $11 = 5 + 5 + 1$

示例 2:

输入: `coins = [2]`, `amount = 3`
 输出: -1

Idea:

```
coins = [1, 2, 5], amount = 11
```

则，当 $i == 0$ 时无法用硬币组成，为 0。当 $i < 0$ 时，忽略 $F(i)$

F(i)	最小硬币数量
F(0)	0 //金额为0不能由硬币组成
F(1)	1 // $F(1) = \min(F(1-1), F(1-2), F(1-5)) + 1 = 1$
F(2)	1 // $F(2) = \min(F(2-1), F(2-2), F(2-5)) + 1 = 1$
F(3)	2 // $F(3) = \min(F(3-1), F(3-2), F(3-5)) + 1 = 2$
F(4)	2 // $F(4) = \min(F(4-1), F(4-2), F(4-5)) + 1 = 2$
...	...
F(11)	3 // $F(11) = \min(F(11-1), F(11-2), F(11-5)) + 1 = 3$

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int max = amount+1;
        int[] dp = new int[amount+1];

        Arrays.fill(dp,max);
        dp[0] = 0;
        for(int i=1;i<=amount;i++){
            for(int j=0;j<coins.length;j++){
                if(coins[j]<=i){
                    dp[i] = Math.min(dp[i],dp[i-coins[j]]+1);
                }
            }
        }

        return dp[amount]>amount ? -1:dp[amount];
    }
}
```

day09 (买卖股票系列)

121. 买卖股票的最佳时机

难度 简单

👍 816



给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

输入: [7,1,5,3,6,4]

输出: 5

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = $6 - 1 = 5$ 。

注意利润不能是 $7 - 1 = 6$ ，因为卖出价格需要大于买入价格。

示例 2:

输入: [7,6,4,3,1]

输出: 0

解释: 在这种情况下，没有交易完成，所以最大利润为 0。

```
class Solution {
    public int maxProfit(int[] prices) {
        //用minPriceBefore记录截止当前天的最低历史价格
        int minPriceBefore = Integer.MAX_VALUE;
        int maxProfit = 0;
        for(int i=0;i<prices.length;i++){
            if(prices[i]<minPriceBefore){
                minPriceBefore = prices[i];
            }else if(prices[i]-minPriceBefore>maxProfit){
                maxProfit = prices[i]-minPriceBefore;
            }
        }
        return maxProfit;
    }
}
```

122. 买卖股票的最佳时机 II

难度 简单  621     

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

输入: `[7,1,5,3,6,4]`

输出: 7

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = $6 - 3 = 3$ 。

示例 2:

输入: `[1,2,3,4,5]`

输出: 4

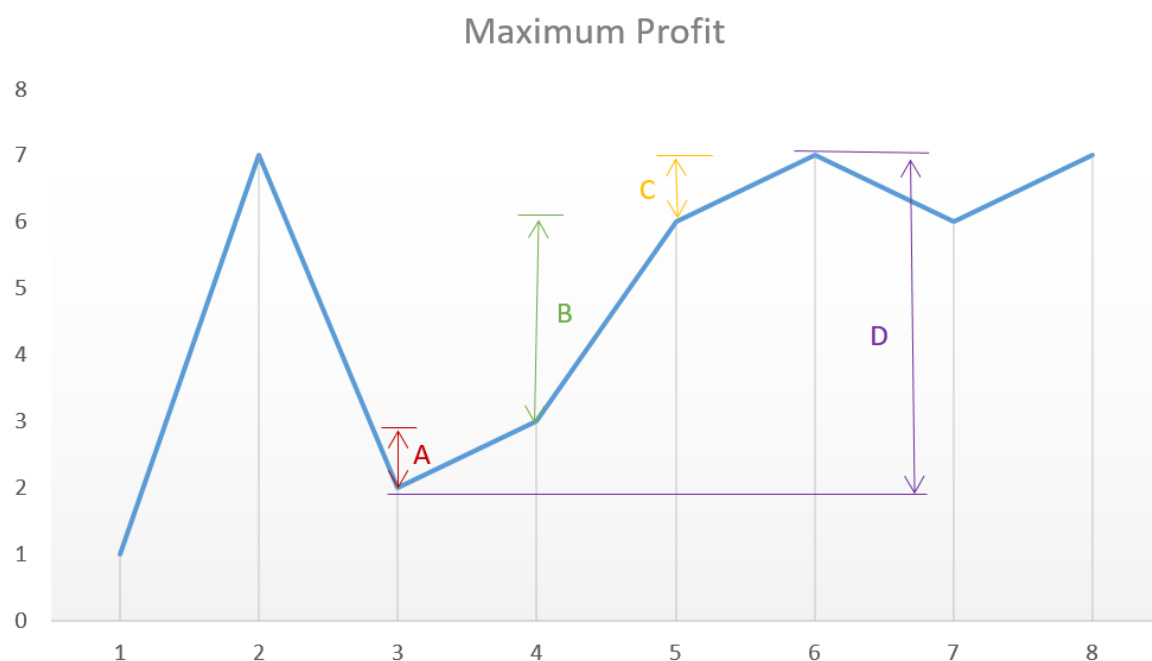
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。

因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

我们可以直接继续增加数组的连续数字之间的差值，如果第二个数字大于第一个数字，我们获得的总和将是最大利润。这种方法将简化解决方案。

`[1,7,2,3,6,7,6,7]`



```

class Solution {
    public int maxProfit(int[] prices) {
        int maxprofit = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1])
                maxprofit += prices[i] - prices[i - 1];
        }
        return maxprofit;
    }
}

```

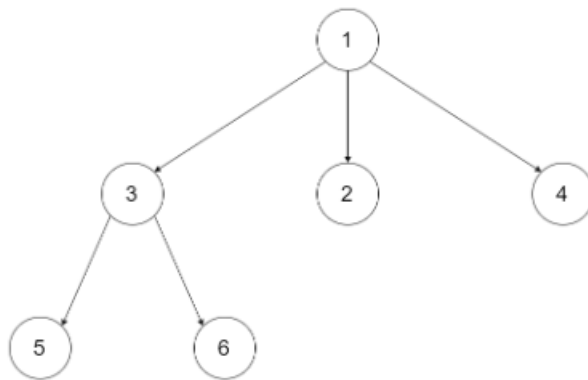
day10 (树的遍历系列)

589. N叉树的前序遍历

难度 简单  66     

给定一个 N 叉树，返回其节点值的 *前序遍历*。

例如，给定一个 3叉树：



返回其前序遍历: [1,3,5,6,2,4]。

递归:

```

/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};

```

```

*/
//前序遍历
class Solution {
    List<Integer> NodeList = new LinkedList<>();
    public List<Integer> preorder(Node root) {
        if(root==null){
            return NodeList;
        }

        NodeList.add(root.val);
        for(Node child:root.children){
            preorder(child);
        }
        return NodeList;
    }
}

//后序遍历
class Solution {
    List<Integer> ans = new LinkedList<Integer>();
    public List<Integer> postorder(Node root) {
        if(root == null){
            return ans;
        }
        for(Node child:root.children){
            postorder(child);
        }
        ans.add(root.val);
        return ans;
    }
}

//非递归--前序遍历
class Solution {
    public List<Integer> preorder(Node root) {
        List<Integer> stack = new LinkedList<>();
        List<Integer> output = new LinkedList<>();
        if(root == null){
            return output;
        }

        stack.add(root.val);
        while(!stack.isEmpty()){
            Node node = stack.pollLast();
            output.add(node.val);
            Collections.reverse(node.children);
            for(Node temp:node.children){
                stack.add(temp);
            }
        }
        return output;
    }
}

```

543. 二叉树的直径

难度 简单

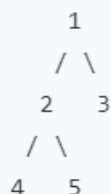
263



给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过根结点。

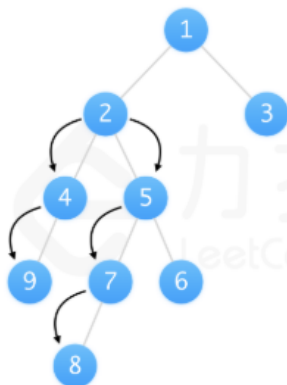
示例：

给定二叉树



返回 3，它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。



如图我们可以知道路径 [9, 4, 2, 5, 7, 8] 可以被看作以 2 为起点，从其左儿子向下遍历的路径 [2, 4, 9] 和从其右儿子向下遍历的路径 [2, 5, 7, 8] 拼接得到。

假设我们知道对于该节点的左儿子向下遍历经过最多的节点数 L （即以左儿子为根的子树的深度）和其右儿子向下遍历经过最多的节点数 R （即以右儿子为根的子树的深度），那么以该节点为起点的路径经过节点数的最大值即为 $L + R + 1$ 。

我们记节点 $node$ 为起点的路径经过节点数的最大值为 d_{node} ，那么二叉树的直径就是所有节点 d_{node} 的最大值减一。

最后的算法流程为：我们定义一个递归函数 `depth(node)` 计算 d_{node} ，函数返回该节点为根的子树的深度。先递归调用左儿子和右儿子求得它们为根的子树的深度 L 和 R ，则该节点为根的子树的深度即为

$$\max(L, R) + 1$$

该节点的 d_{node} 值为

$$L + R + 1$$

递归搜索每个节点并设一个全局变量 ans 记录 d_{node} 的最大值，最后返回 `ans-1` 即为树的直径。

```
class Solution {
    int ans;
    public int diameterOfBinaryTree(TreeNode root) {
        ans = 1;
        depth(root);
        return ans - 1;
    }
    public int depth(TreeNode node) {
```



```

        if (node == null) return 0; // 访问到空节点了，返回0
        int L = depth(node.left); // 左儿子为根的子树的深度
        int R = depth(node.right); // 右儿子为根的子树的深度
        ans = Math.max(ans, L+R+1); // 计算d_node即L+R+1 并更新ans
        return Math.max(L, R) + 1; // 返回该节点为根的子树的深度
    }
}

```

day11（将数组分成和相等的三部分）

1013. 将数组分成和相等的三个部分

难度 简单  64     

给你一个整数数组 `A`，只有可以将其划分为三个和相等的非空部分时才返回 `true`，否则返回 `false`。

形式上，如果可以找出索引 $i+1 < j$ 且满足 $(A[0] + A[1] + \dots + A[i] == A[i+1] + A[i+2] + \dots + A[j-1] == A[j] + A[j+1] + \dots + A[A.length - 1])$ 就可以将数组三等分。

示例 1：

输出：[0,2,1,-6,6,-7,9,1,2,0,1]

输出：true

解释： $0 + 2 + 1 = -6 + 6 - 7 + 9 + 1 = 2 + 0 + 1$

示例 2：

输入：[0,2,1,-6,6,7,9,-1,2,0,1]

输出：false

示例 3：

输入：[3,3,6,5,-2,2,5,1,-9,4]

输出：true

解释： $3 + 3 = 6 = 5 - 2 + 2 + 5 + 1 - 9 + 4$

```

class Solution {
    public boolean canThreePartsEqualSum(int[] A) {
        int sum = 0;
        for(int i:A){
            sum+=i;
        }
        if(sum%3!=0){
            return false;
        }

        int s = 0;
        int flag = 0;
        for(int i=0;i<A.length;i++){
            s+=A[i];
            if(sum/3==s){

```

```

        flag++;
        s = 0;
    }
}
return flag>=3; //防止[10,-10,10,-10,10,-10,10,-10] sum=0 可能导致flag>3
}
}

```

day12 (字符串的最大公因子)

1071. 字符串的最大公因子

难度 简单  96     

对于字符串 S 和 T ，只有在 $S = T + \dots + T$ (T 与自身连接 1 次或多次) 时，我们才认定“ T 能除尽 S ”。

返回最长字符串 X ，要求满足 X 能除尽 $str1$ 且 X 能除尽 $str2$ 。

示例 1:

输入: $str1 = \text{"ABCABC"}, str2 = \text{"ABC"}$
输出: "ABC"

示例 2:

输入: $str1 = \text{"ABABAB"}, str2 = \text{"ABAB"}$
输出: "AB"

示例 3:

输入: $str1 = \text{"LEET"}, str2 = \text{"CODE"}$
输出: ""

```

class Solution {

    public int gcd(int len1,int len2){
        int big = len1>len2 ? len1:len2;
        int small = len1<len2 ? len1:len2;
        if(big%small==0){
            return small;
        }
        return gcd(big%small,small);
    }
    public String gcdOfStrings(String str1, String str2) {
        if(!(str1+str2).equals(str2+str1)){
            return "";
        }
        int gcdStr = gcd(str1.length(),str2.length());
        return str1.substring(0,gcdStr);
    }
}

```

```
}
```

day13 (多数元素)

169. 多数元素

难度 简单

👍 505



给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

输入: [3,2,3]

输出: 3

示例 2:

输入: [2,2,1,1,1,2,2]

输出: 2

```
class Solution {
    public int majorityElement(int[] nums) {
        Map<Integer,Integer> record = new HashMap<>();
        int n = nums.length;
        for(int i=0;i<n;i++){
            if(!record.containsKey(nums[i])){
                record.put(nums[i],1);
            }else{
                record.put(nums[i],record.get(nums[i])+1);
            }
        }
        int maxValue = 0;
        int ans = 0;
        for(Map.Entry<Integer,Integer> entry:record.entrySet()){
            if(entry.getValue()>maxValue){
                maxValue = entry.getValue();
                ans = entry.getKey();
            }
        }
        return ans;
    }
}
```

遍历Map的四种方式

```
public static void main(String[] args) {
    // 循环遍历Map的4中方法
    Map<Integer,Integer> map = new HashMap<Integer,Integer>();
    map.put(1,2);
    // 1. entrySet遍历，在键和值都需要时使用（最常用）
    for (Map.Entry<Integer,Integer> entry : map.entrySet()) {
```

```

        System.out.println("key = " + entry.getKey() + ", value = " +
entry.getValue());
    }
    // 2. 通过keySet或values来实现遍历,性能略低于第一种方式
    // 遍历map中的键
    for (Integer key : map.keySet()) {
        System.out.println("key = " + key);
    }
    // 遍历map中的值
    for (Integer value : map.values()) {
        System.out.println("key = " + value);
    }
    // 3. 使用Iterator遍历
    Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
    while (it.hasNext()) {
        Map.Entry<Integer, Integer> entry = it.next();
        System.out.println("key = " + entry.getKey() + ", value = " +
entry.getValue());
    }

    // 4. java8 Lambda
    // java8提供了Lambda表达式支持,语法看起来更简洁,可以同时拿到key和value,
    // 不过,经测试,性能低于entrySet,所以更推荐用entrySet的方式
    map.forEach((key, value) -> {
        System.out.println(key + ":" + value);
    });
}

```

day14 (最长上升子序列--动态规划)

300. 最长上升子序列

难度 中等  524     

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例：

输入：[10,9,2,5,3,7,101,18]

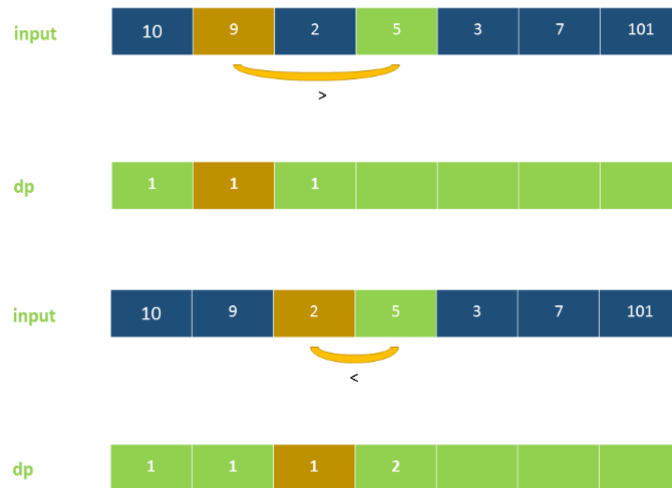
输出：4

解释：最长的上升子序列是 [2,3,7,101]，它的长度是 4。

说明：

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 $O(n^2)$ 。

思路：



```
class Solution {
    public int lengthOfLIS(int[] nums) {
        //动态规划
        if(nums.length==0){
            return 0;
        }
        //dp[i]代表数组中第<=i的子数组中最长上升子序列长度
        int[] dp = new int[nums.length];
        int maxAns = 1;
        for(int i=0;i<nums.length;i++){
            int maxBeforeI = 0; //定义dp[<i]中最大的值，即子数组最长上升子序列长度
            dp[i] = 1;
            for(int j=0;j<i;j++){
                //找到nums[i]>nums[j]时候 dp[1..j]中的最大值
                //只有nums[i]>nums[i]的时候，才会出现最长生长子序列递增，才去更新dp
                if(nums[i]>nums[j]){
                    maxBeforeI = Math.max(maxBeforeI,dp[j]);
                }
            }
            dp[i] = maxBeforeI+1;
            maxAns = Math.max(maxAns,dp[i]);
        }
        return maxAns;
    }
}
```

动态规划类似题目：

334. 递增的三元子序列

难度 中等

115



给定一个未排序的数组，判断这个数组中是否存在长度为 3 的递增子序列。

数学表达式如下：

如果存在这样的 i, j, k ，且满足 $0 \leq i < j < k \leq n-1$ ，
使得 $arr[i] < arr[j] < arr[k]$ ，返回 true；否则返回 false。

说明：要求算法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

示例 1:

输入：[1,2,3,4,5]

输出：true

示例 2:

输入：[5,4,3,2,1]

输出：false

```
class Solution {
    public boolean increasingTriplet(int[] nums) {
        if(nums.length<3){
            return false;
        }
        int[] dp = new int[nums.length];
        for(int i=0;i<nums.length;i++){
            int maxBeforeI = 0;
            for(int j=0;j<i;j++){
                if(nums[i]>nums[j]){
                    maxBeforeI = Math.max(maxBeforeI,dp[j]);
                }
            }
            dp[i] = maxBeforeI + 1;
        }
        for(int i:dp){
            if(i>=3){
                return true;
            }
        }
        return false;
    }
}
```

53. 最大子序和

难度 简单

👍 1716



给定一个整数数组 `nums`，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

输入: `[-2,1,-3,4,-1,2,1,-5,4]`,

输出: 6

解释: 连续子数组 `[4,-1,2,1]` 的和最大, 为 6。

```
class Solution {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        for(int i=1;i<nums.length;i++){
            if(dp[i-1]>0){
                dp[i] = dp[i-1]+nums[i];
            }else{
                dp[i] = nums[i];
            }
            maxSum = Math.max(maxSum,dp[i]);
        }
        return maxSum;
    }
}
```

day15 (岛屿的最大面积DFS)

695. 岛屿的最大面积

难度 中等

👍 216



给定一个包含了一些 0 和 1 的非空二维数组 `grid`，一个 岛屿 是由四个方向（水平或垂直）的 1（代表土地）构成的组合。你可以假设二维矩阵的四个边缘都被水包围着。

找到给定的二维数组中最大的岛屿面积。（如果没有岛屿，则返回面积为 0。）

示例 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,0,0,0,0,0,0,0,1,0,0],
 [0,0,0,0,0,0,0,0,1,1,0,0,0],
 [0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

对于上面这个给定矩阵应返回 6。注意答案不应该是11，因为岛屿只能包含水平或垂直的四个方向的‘1’。

示例 2:

```
[[0,0,0,0,0,0,0,0,0]]
```

```
class Solution {
    public int maxAreaOfIsland(int[][] grid) {
        int maxArea = 0;
        for(int i=0;i<grid.length;i++){
            for(int j=0;j<grid[i].length;j++){
                if(grid[i][j]==1){
                    maxArea = Math.max(maxArea,dfs(i,j,grid));
                }
            }
        }
        return maxArea;
    }
    //递归遍历寻找最大连续1的个数（岛屿面积）
    public int dfs(int i,int j,int[][] grid){
        if(i<0 || j<0 || i>=grid.length || j>=grid[i].length || grid[i][j]==0){
            return 0;
        }
        grid[i][j]=0;//把当前[i][j]置为0 防止回溯遍历出现栈溢出
        int num = 1;
        num += dfs(i-1,j,grid);
        num += dfs(i+1,j,grid);
        num += dfs(i,j-1,grid);
        num += dfs(i,j+1,grid);
        return num;
    }
}
```



```

//采用栈的方式
class Solution {

    public int maxAreaOfIsland(int[][] grid) {
        Deque<int[]> stack = new LinkedList<>();

        int[][] moveIndexArray = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        int maxArea = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                stack.add(new int[]{i, j});
                //计算最大面积
                int currMaxArea = 0;
                while (!stack.isEmpty()) {
                    int[] pop = stack.pop();
                    int currI = pop[0];
                    int currJ = pop[1];
                    if (currI < 0 || currI >= grid.length || currJ < 0 || currJ
>= grid[0].length || grid[currI][currJ] == 0) {
                        continue;
                    }
                    currMaxArea++;
                    grid[currI][currJ] = 0;
                    for (int[] moveIndex : moveIndexArray) {
                        stack.add(new int[]{currI + moveIndex[0], currJ +
moveIndex[1]});
                    }
                }
                maxArea = Math.max(currMaxArea, maxArea);
            }
        }

        return maxArea;
    }
}

```

day16 (字符串压缩--双指针)

面试题 01.06. 字符串压缩

难度 简单

👍 29



字符串压缩。利用字符重复出现的次数，编写一种方法，实现基本的字符串压缩功能。比如，字符串 `aabcccccaaa` 会变为 `a2b1c5a3`。若“压缩”后的字符串没有变短，则返回原先的字符串。你可以假设字符串中只包含大小写英文字母（a至z）。

示例1:

输入: "aabcccccaaa"

输出: "a2b1c5a3"

示例2:

输入: "abbccd"

输出: "abbccd"

解释: "abbccd"压缩后为"a1b2c2d1"，比原字符串长度更长。

```
class Solution {
    public String compressString(String S) {
        int N = S.length();
        int i=0;
        StringBuilder sb = new StringBuilder();
        while(i<N){
            int j=i;
            while(j<N && S.charAt(j)==S.charAt(i)){
                j++;
            }
            sb.append(S.charAt(i)).append(j-i);
            i = j;
        }

        String ans = sb.toString();
        return ans.length()>=N ? S:ans;
    }
}
```

day17 (拼写单词)

1160. 拼写单词

难度 简单

74



给你一份『词汇表』（字符串数组）`words` 和一张『字母表』（字符串）`chars`。

假如你可以用 `chars` 中的『字母』（字符）拼写出 `words` 中的某个『单词』（字符串），那么我们就认为你掌握了这个单词。

注意：每次拼写时，`chars` 中的每个字母都只能用一次。

返回词汇表 `words` 中你掌握的所有单词的 长度之和。

示例 1：

输入：words = ["cat","bt","hat","tree"], chars = "atach"

输出：6

解释：

可以形成字符串 "cat" 和 "hat"，所以答案是 3 + 3 = 6。

示例 2：

输入：words = ["hello","world","leetcode"], chars =

"welldonehoneyr"

输出：10

解释：

可以形成字符串 "hello" 和 "world"，所以答案是 5 + 5 = 10。

```
class Solution {
    public int countCharacters(String[] words, String chars) {
        int[] hash = new int[26];
        for(char c:chars.toCharArray()){
            hash[c-'a'] += 1;
        }

        int[] map = new int[26];
        int len = 0;
        for(int i=0;i<words.length;i++){
            String word = words[i];
            Arrays.fill(map,0);
            boolean flag = true;
            for(char c:words[i].toCharArray()){
                map[c-'a'] ++;
                if(map[c-'a']>hash[c-'a']){
                    flag = false;
                }
            }
            if(flag){
                len+=word.length();
            }else{
                len+=0;
            }
        }
    }
}
```

```

    }
    return len;
}
}

```

day18 (矩形重叠)

836. 矩形重叠

难度 简单  108     

矩形以列表 `[x1, y1, x2, y2]` 的形式表示，其中 `(x1, y1)` 为左下角的坐标，`(x2, y2)` 是右上角的坐标。

如果相交的面积为正，则称两矩形重叠。需要明确的是，只在角或边接触的两个矩形不构成重叠。

给出两个矩形，判断它们是否重叠并返回结果。

示例 1:

```

输入: rec1 = [0,0,2,2], rec2 = [1,1,3,3]
输出: true

```

示例 2:

```

输入: rec1 = [0,0,1,1], rec2 = [1,0,2,1]
输出: false

```

//想象一下，如果我们在平面中放置一个固定的矩形 `rec2`，那么矩形 `rec1` 必须要出现在 `rec2` 的「四周」，也就是说，矩形 `rec1` 需要满足以下四种情况中的至少一种：

//矩形 `rec1` 在矩形 `rec2` 的左侧；

//矩形 `rec1` 在矩形 `rec2` 的右侧；

//矩形 `rec1` 在矩形 `rec2` 的上方；

//矩形 `rec1` 在矩形 `rec2` 的下方。

```

class Solution {
    public boolean isRectangleOverlap(int[] rec1, int[] rec2) {
        //先判断不重叠的情况 再取反
        return (!(rec1[1]>=rec2[3]||rec1[3]<=rec2[1]||rec1[2]
<=rec2[0]||rec1[0]>=rec2[2]));
    }
}

```

day19 (最长回文串)

409. 最长回文串

难度 简单

👍 143



给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造的最长的回文串。

在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

注意：

假设字符串的长度不会超过 1010。

示例 1:

输入：
"abcccccdd"

输出：
7

解释：
我们可以构造的最长的回文串是"dccaccd"，它的长度是 7。

对于每个字符 `ch`，假设它出现了 `v` 次，我们可以使用该字符 `v / 2 * 2` 次，在回文串的左侧和右侧分别放置 `v / 2` 个字符 `ch`，其中 `/` 为整数除法。例如若 "a" 出现了 5 次，那么我们可以使用 "a" 的次数为 4，回文串的左右两侧分别放置 2 个 "a"。

如果有任何一个字符 `ch` 的出现次数 `v` 为奇数（即 `v % 2 == 1`），那么可以将这个字符作为回文中心，注意只能最多有一个字符作为回文中心。在代码中，我们用 `ans` 存储回文串的长度，由于在遍历字符时，`ans` 每次会增加 `v / 2 * 2`，因此 `ans` 一直为偶数。但在发现了第一个出现次数为奇数的字符后，我们将 `ans` 增加 1，这样 `ans` 变为奇数，在后面发现其它出现奇数次的字符时，我们就不改变 `ans` 的值了。

```
class Solution {
    public int longestPalindrome(String s) {
        int[] arr = new int[128];
        for(char c:s.toCharArray()){
            arr[c]++;
        }

        int ans = 0;
        for(int n:arr){
            ans+=n/2*2;
            //第一次遇到奇数字符时，ans+1,以后遇到奇数不再增加ans直接跳过，只有偶数增加在
            //ans左右两端
            //ans%2==0 防止以后遍历再次进入if语句
            if(n%2==1 && ans%2==0 ){
                ans ++;
            }
        }
        return ans;
    }
}
```

```
}  
}
```

day20 (最小的k个数)

面试题40. 最小的k个数

难度 简单

👍 53



输入整数数组 `arr`，找出其中最小的 `k` 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1:

输入: `arr = [3,2,1]`, `k = 2`
输出: `[1,2]` 或者 `[2,1]`

示例 2:

输入: `arr = [0,1,2,1]`, `k = 1`
输出: `[0]`

相等于练习冒泡排序吧 哈哈

```
class Solution {  
    public int[] getLeastNumbers(int[] arr, int k) {  
        sortArray(arr);  
        int[] ans = new int[k];  
        for(int i=0;i<k;i++){  
            ans[i] = arr[i];  
        }  
        return ans;  
    }  
  
    public void sortArray(int[] arr){  
        for(int i=0;i<arr.length;i++){  
            boolean flag = true;  
            for(int j=0;j<arr.length-i-1;j++){  
                if(arr[j]>arr[j+1]){  
                    flag = false;  
                    int temp = arr[j];  
                    arr[j] = arr[j+1];  
                    arr[j+1] = temp;  
                }  
            }  
            if(flag){  
                break;  
            }  
        }  
    }  
}
```

day21 (水壶问题--最大公约数)

365. 水壶问题

难度 中等

157



有两个容量分别为 x 升 和 y 升 的水壶以及无限多的水。请判断能否通过使用这两个水壶，从而可以得到恰好 z 升 的水？

如果可以，最后请用以上水壶中的一或两个来盛放取得的 z 升 水。

你允许：

- 装满任意一个水壶
- 清空任意一个水壶
- 从一个水壶向另外一个水壶倒水，直到装满或者倒空

示例 1: (From the famous "*Die Hard*" example)

输入: $x = 3, y = 5, z = 4$

输出: True

示例 2:

输入: $x = 2, y = 6, z = 5$

输出: False

idea:

若 a, b 是整数,且 $\gcd(a, b)=d$, 那么对于任意的整数 $x, y, ax+by$ 都一定是 d 的倍数, 特别地, 一定存在整数 x, y , 使 $ax+by=d$ 成立。

* 裴蜀定理:

* 如果所需要的水量是两个水壶容量的最大公约数的倍数, 且水量不大于两个水壶的容量之和, 那么必然可以用这两个水壶操作得到所需要的水量。

```
class Solution {
    public boolean canMeasureWater(int x, int y, int z) {
        //解决x或者y中一个为0的情况
        if(x==0 || y==0){
            if(z==x || z==y){
                return true;
            }
            return false;
        }
        if(x+y<z){
            return false;
        }
        int temp = gcd(x,y);
        return z%temp==0;
    }

    private int gcd(int x,int y){
        int big = x>y? x:y;
        int small = x<y? x:y;
        if(big%small==0){
```

```

        return small;
    }
    return gcd(big%small,small);
}
}

```

day22 (使数组唯一的最小增量)

945. 使数组唯一的最小增量

难度 中等  98     

给定整数数组 A，每次 *move* 操作将会选择任意 $A[i]$ ，并将其递增 1。

返回使 A 中的每个值都是唯一的最少操作次数。

示例 1:

输入: [1,2,2]
 输出: 1
 解释: 经过一次 *move* 操作，数组将变为 [1, 2, 3]。

示例 2:

输入: [3,2,1,2,1,7]
 输出: 6
 解释: 经过 6 次 *move* 操作，数组将变为 [3, 4, 1, 2, 5, 7]。
 可以看出 5 次或 5 次以下的 *move* 操作是不能让数组的每个值唯一的。

提示:

1. $0 \leq A.length \leq 40000$
2. $0 \leq A[i] < 40000$

```

class Solution {
    public int minIncrementForUnique(int[] A) {
        //先排序
        Arrays.sort(A);
        int ans = 0;
        for(int i=1;i<A.length;i++){
            if(A[i]<=A[i-1]){
                int temp = A[i];
                A[i] = A[i-1]+1;
                ans+=A[i-1]+1-temp;
            }
        }
        return ans;
    }
}

```

day23 (链表中间节点--快慢指针)

876. 链表的中间结点

难度 简单  186     

给定一个带有头结点 `head` 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

示例 1:

输入: [1,2,3,4,5]

输出: 此列表中的结点 3 (序列化形式: [3,4,5])

返回的结点值为 3 。(测评系统对该结点序列化表述是 [3,4,5])。

注意，我们返回了一个 `ListNode` 类型的对象 `ans`，这样：

`ans.val = 3`, `ans.next.val = 4`, `ans.next.next.val = 5`, 以及
`ans.next.next.next = NULL`。

示例 2:

输入: [1,2,3,4,5,6]

输出: 此列表中的结点 4 (序列化形式: [4,5,6])

由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode middleNode(ListNode head) {
        if(head==null){
            return null;
        }
        ListNode fast = head;
        ListNode slow = head;
        while(fast!=null && fast.next!=null){
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

day24 (按摩师问题--动态规划)

面试题 17.16. 按摩师

难度 简单

73



一个有名的按摩师会收到源源不断的预约请求，每个预约都可以选择接或不接。在每次预约服务之间要有休息时间，因此她不能接受相邻的预约。给定一个预约请求序列，替按摩师找到最优的预约集合（总预约时间最长），返回总的分钟数。

注意：本题相对原题稍作改动

示例 1：

输入：[1,2,3,1]

输出：4

解释：选择 1 号预约和 3 号预约，总时长 = 1 + 3 = 4。

示例 2：

输入：[2,7,9,3,1]

输出：12

解释：选择 1 号预约、3 号预约和 5 号预约，总时长 = 2 + 9 + 1 = 12。

示例 3：

输入：[2,1,4,5,3,1,1,3]

输出：12

解释：选择 1 号预约、3 号预约、5 号预约和 8 号预约，总时长 = 2 + 4 + 3 + 3 = 12。

idea:

$dp[i][0]$ 第 i 个预约不接受的最佳总时长，则 $i-1$ 可以接收也可以不接受

$dp[i][1]$ 第 i 个预约接收的最佳总时长，则 $i-1$ 一定为不接受

```
class Solution {
    public int massage(int[] nums) {
        int n = nums.length;
        if(n==0) return 0;
        if(n==1) return nums[0];

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

day25 (三维形体表面积)

892. 三维形体的表面积

难度 简单  93     

在 $N * N$ 的网格上，我们放置一些 $1 * 1 * 1$ 的立方体。

每个值 $v = \text{grid}[i][j]$ 表示 v 个正方体叠放在对应单元格 (i, j) 上。

请你返回最终形体的表面积。

示例 1:

输入: $[[2]]$

输出: 10

示例 2:

输入: $[[1,2],[3,4]]$

输出: 34

示例 3:

输入: $[[1,0],[0,2]]$

输出: 16

示例 4:

输入: $[[1,1,1],[1,0,1],[1,1,1]]$

输出: 32

Idea:

- 1、输入二维数组，数字代表数组*i,j*位置放置的立方体个数
- 2、要点是要把*i,j*方向上立方体重叠面积进行删除

```
class Solution {
    public int surfaceArea(int[][] grid) {
        int n = grid.length;
        int area = 0;
        for(int i = 0; i < n; i++){
            for(int j = 0; j < grid[i].length; j++){
                int level = grid[i][j];
                if(level > 0){
                    area += (level << 2) + 2;
                }
                //删除i,j方向上的重叠面积
                if(i > 0) area -= Math.min(grid[i-1][j], grid[i][j]) * 2;
                if(j > 0) area -= Math.min(grid[i][j-1], grid[i][j]) * 2;
            }
        }
    }
}
```

```

    }
    return area;
}
}

```

day26 (车的可用捕获量)

999. 车的可用捕获量

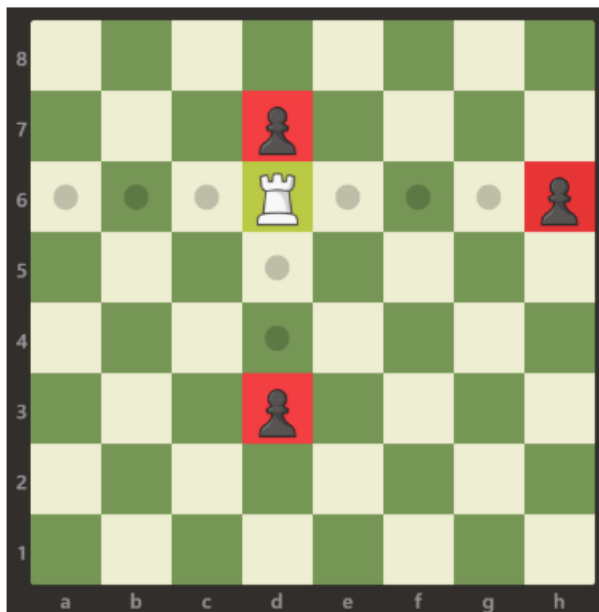
难度 简单  60     

在一个 8×8 的棋盘上，有一个白色车（rook）。也可能有空方块，白色的象（bishop）和黑色的卒（pawn）。它们分别以字符 “R”，“.”，“B” 和 “p” 给出。大写字符表示白棋，小写字符表示黑棋。

车按国际象棋中的规则移动：它选择四个基本方向中的一个（北，东，西和南），然后朝那个方向移动，直到它选择停止、到达棋盘的边缘或移动到同一方格来捕获该方格上颜色相反的卒。另外，车不能与其他友方（白色）象进入同一个方格。

返回车能够在一次移动中捕获到的卒的数量。

示例 1：



输入：[[".", ".", ".", ".", ".", ".", ".", "."],
 [".", ".", ".", "p", ".", ".", ".", "."], [".", ".", ".", "R", ".", ".", ".", "p"],
 [".", ".", ".", ".", ".", ".", ".", "."], [".", ".", ".", ".", ".", ".", ".", "."],
 [".", ".", ".", "p", ".", ".", ".", "."], [".", ".", ".", ".", ".", ".", ".", "."],
 [".", ".", ".", ".", ".", ".", ".", "."]]

输出：3

解释：

在本例中，车能够捕获所有的卒。

idea:

题目太长迷惑人心，要点如下：

- 1、先找到白色车的位置 i, j
- 2、在四个方向上移动车，车停止条件为 a: 超出 8×8 棋盘界限 b: 遇到白色的象 c: 遇到要找的黑色卒

```

class Solution {
    public int numRookCaptures(char[][] board) {
        //上下左右
        int[] dx = {-1,1,0,0};
        int[] dy = {0,0,-1,1};

        int ans = 0;
        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                //找到白色车的位置
                if(board[i][j]=='R'){
                    //四个方向进行前进
                    for(int k=0;k<4;k++){
                        //把白色车的位置赋值给移动变量
                        int moveX = i;
                        int moveY = j;
                        while(true){
                            moveX += dx[k];
                            moveY += dy[k];
                            //判断是否越界 车不能与其他友方（白色）象进入同一个方
                            if(moveX<0 || moveX>=8 || moveY<0 || moveY>=8 ||
board[moveX][moveY]=='B'){
                                break;
                            }

                            if(board[moveX][moveY]=='p'){
                                ans++;
                                break; //在另一个方向查找
                            }
                        }
                    }
                }
            }
        }
        return ans;
    }
}

```

day27（卡牌分组）

914. 卡牌分组

难度 简单

👍 133



给定一副牌，每张牌上都写着一个整数。

此时，你需要选定一个数字 X ，使我们可以将整副牌按下述规则分成 1 组或更多组：

- 每组都有 X 张牌。
- 组内所有的牌上都写着相同的整数。

仅当你可选的 $X \geq 2$ 时返回 `true`。

示例 1：

输入：[1,2,3,4,4,3,2,1]

输出：true

解释：可行的分组是 [1,1], [2,2], [3,3], [4,4]

示例 2：

输入：[1,1,1,2,2,2,3,3]

输出：false

解释：没有满足要求的分组。

示例 3：

输入：[1]

输出：false

解释：没有满足要求的分组。

idea:

求数组中各个相同元素个数的最大公约数，如果最大公约数 ≥ 2 则返回true 否则返回false

```
class Solution {
    public boolean hasGroupsSizeX(int[] deck) {
        int[] counter = new int[10000];
        for(int i=0;i<deck.length;i++){
            counter[deck[i]] ++;
        }
        int x =0;
        for(int count:counter){
            if(count>0){
                //计算最大公约数
                x = gcd(x,count);
                if(x==1){
                    return false;
                }
            }
        }
        return x>=2;
    }
}
```

```

private int gcd(int a,int b){
    if(a==0) return b;
    if(b==0) return a;
    int max = a>b? a:b;
    int min = a<b? a:b;
    if(max%min==0){
        return min;
    }
    return gcd(max%min,min);
}
}

```

BFS/DFS问题

快速幂算法

```

public static long pow(int n) {
    if (n == 0)
        return 1;
    long half = pow(n / 2);
    if (n % 2 == 0)
        return (half * half);
    else
        return (half * half * 2);
}

```

BFS地图问题（阿里巴巴面试题）

一个地图 $n*m$ ，包含1个起点，1个终点，其他点包括可达点和不可达点。每一次可以：上下左右移动，或使用1点能量从 (i,j) 瞬间移动到 $(n-1-i, m-1-j)$ ，最多可以使用5点能量。

```

package cn.nuaa.alibaba;

import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class test02 {
    static int[] dx = {1,-1,0,0};
    static int[] dy = {0,0,1,-1};
    static int m;
    static int n;
    static int endX;
    static int endY;
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        m = scanner.nextInt();
        char[][] map = new char[n][m];
        Queue<Pair> queue = new LinkedList<>();
        for(int i=0;i<n;i++){
            map[i] = scanner.next().toCharArray();
            for(int j=0;j<map[i].length;j++){
                if(map[i][j]=='S'){

```

```

        Pair pair = new Pair(i,j);
        queue.add(pair);
    }else if(map[i][j]=='E'){
        endX = i;
        endY = j;
    }
}
}
//BFS
System.out.println(BFS(map,queue));
}
public static boolean check(int x,int y){
    if(x<0 || y<0 || x>=n || y>=m){
        return false;
    }
    return true;
}
public static int BFS(char[][] map,Queue<Pair>queue){
    while (!queue.isEmpty()){
        int size = queue.size();
        while (size-- >0){
            Pair top = queue.poll();
            if(top.x==endX && top.y==endY){
                return top.step;
            }
            for(int k=0;k<4;k++){
                int curX = top.x+dx[k];
                int curY = top.y+dy[k];
                Pair nextPair = new Pair(curX,curY);
                nextPair.step = top.step + 1;
                nextPair.fly = top.fly;
                if(check(curX,curY) && (map[curX][curY]=='.' || map[curX]
[curY]=='E')){
                    queue.add(nextPair);
                    map[curX][curY] = 'X';
                }
            }
            int flyX = n-1-top.x;
            int flyY = m-1-top.y;
            if(check(flyX,flyY) && top.fly<5 && (map[flyX][flyY]=='.' ||
map[flyX][flyY]=='E')){
                Pair pair = new Pair(flyX,flyY);
                pair.step = top.step+1;
                pair.fly = top.fly+1;
                queue.add(pair);
                map[flyX][flyY] = 'X';
            }
        }
    }
    return -1;
}
}

class Pair{
    int x;
    int y;
    int step;
    int fly;
}

```



```
public Pair(int x, int y) {  
    this.x = x;  
    this.y = y;  
}  
}
```

LeetCode BFS问题

被围绕的区域

130. 被围绕的区域

难度 中等  181     

给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例：

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

运行你的函数后，矩阵变为：

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

解释：

被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

总结：

- 1、一般需要定义一个内部类，代表地图每个点，将题目中给的属性加进去，基础属性为坐标x,y
- 2、BFS问题需要queue，DFS问题需要stack
- 3、一般需要定义上下左右坐标转移数组
- 4、需要定义坐标范围检查函数

```
class Solution {  
    private class Node{  
        int x;  
        int y;  
        public Node(int x,int y){  
            this.x = x;  
        }  
    }  
}
```

```

        this.y = y;
    }
}

int m = 0;
int n = 0;
public void solve(char[][] board) {
    if(board.length==0 || board==null){
        return;
    }
    m = board.length;
    n = board[0].length;
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            boolean isEdge = (i==0 || i==m-1 || j==0 || j==n-1) ? true:false;
            if(isEdge && board[i][j]=='O'){
                bfs(board,i,j);
            }
        }
    }
}

//与边界相连的O用#代替
for(int i=0;i<m;i++){
    for(int j=0;j<n;j++){
        if(board[i][j] == 'O'){
            board[i][j] = 'X';
        }
        if(board[i][j] == '#'){
            board[i][j] = 'O';
        }
    }
}

public boolean check(int x,int y){
    if(x<0 || x>=m || y<0 || y>=n){
        return false;
    }
    return true;
}

public void bfs(char[][] board,int i,int j){
    Queue<Node> queue = new LinkedList<>();
    queue.add(new Node(i,j));
    board[i][j] = '#';
    //上下左右
    int[] dx = {-1,1,0,0};
    int[] dy = {0,0,-1,1};
    while(!queue.isEmpty()){
        Node curNode = queue.poll();

        for(int k=0;k<4;k++){
            int nextX = curNode.x + dx[k];
            int nextY = curNode.y + dy[k];
            if(check(nextX,nextY) && board[nextX][nextY]=='O'){
                queue.add(new Node(nextX,nextY));
                board[nextX][nextY] = '#';
            }
        }
    }
}

```

```
    }  
  }  
}
```

最远海洋问题

参考day29

动态规划问题

day28 (单词的压缩编码---set存储不重复后缀)

820. 单词的压缩编码

难度 中等 98 收藏 分享 评论

给定一个单词列表，我们将这个列表编码成一个索引字符串 S 与一个索引列表 A 。

例如，如果这个列表是 `["time", "me", "bell"]`，我们就可以将其表示为 $S = "time#bell#"$ 和 $indexes = [0, 2, 5]$ 。

对于每一个索引，我们可以通过从字符串 S 中索引的位置开始读取字符串，直到 `"#"` 结束，来恢复我们之前的单词列表。

那么成功对给定单词列表进行编码的最小字符串长度是多少呢？

示例：

```
输入：words = ["time", "me", "bell"]  
输出：10  
说明：S = "time#bell#" , indexes = [0, 2, 5] 。
```

提示：

1. $1 \leq words.length \leq 2000$
2. $1 \leq words[i].length \leq 7$
3. 每个单词都是小写字母。

idea:

如果单词 X 是 Y 的后缀，那么单词 X 就不需要考虑了，因为编码 Y 的时候就同时将 X 编码了。例如，如果 $words$ 中同时有 `"me"` 和 `"time"`，我们就可以在不改变答案的情况下不考虑 `"me"`。

如果单词 Y 不在任何别的单词 X 的后缀中出现，那么 Y 一定是编码字符串的一部分。

因此，目标就是保留所有不是其他单词后缀的单词，最后的结果就是这些单词长度加一的总和，因为每个单词编码后后面还需要跟一个 `#` 符号。

```
class Solution {  
    public int minimumLengthEncoding(String[] words) {  
        Set<String> set = new HashSet<String>(Arrays.asList(words));
```

```

除
    for(String word:words){
        for(int i=1;i<word.length();i++){ //i一定从1开始，否则会把该单词从set中移
            set.remove(word.substring(i));
        }
    }

    int ans = 0;
    // for(String word:set){
    //     ans += word.length()+1;
    // }
    Iterator<String> iter = set.iterator();
    while(iter.hasNext()){
        String temp = iter.next();
        ans += temp.length()+1;
    }
    return ans;
}
}

```

day29 (BFS最远海洋问题)

1162. 地图分析

难度 中等  76     

你现在手里有一份大小为 $N \times N$ 的『地图』（网格）`grid`，上面的每个『区域』（单元格）都用 0 和 1 标记好了。其中 0 代表海洋，1 代表陆地，你知道距离陆地区域最远的海洋区域是哪一个吗？请返回该海洋区域到离它最近的陆地区域的距离。

我们这里说的距离是『曼哈顿距离』（Manhattan Distance）： (x_0, y_0) 和 (x_1, y_1) 这两个区域之间的距离是 $|x_0 - x_1| + |y_0 - y_1|$ 。

如果我们的地图上只有陆地或者海洋，请返回 -1。

示例 1：

1	0	1
0	0	0
1	0	1

输入：[[1,0,1],[0,0,0],[1,0,1]]

输出：2

解释：

海洋区域 (1, 1) 和所有陆地区域之间的距离都达到最大，最大距离为 2。

Idea:

我们只要先把所有的陆地都入队，然后从各个陆地**同时开始**一层一层的向海洋扩散，那么最后扩散到的海洋就是最远的海洋！并且这个海洋肯定是被离他最近的陆地给扩散到的！


```

        for(int k=0;k<4;k++){
            int newX = curNode.x+dx[k];
            int newY = curNode.y+dy[k];
            if(check(newX,newY) && grid[newX][newY]==0){
                newNode = new Node(newX,newY);
                newNode.far = curNode.far+1; //一定是curNode.step
                grid[newX][newY] = 1;
                hasOcean = true;
                queue.add(newNode);
            }
        }
    }
    if(!hasOcean){
        return -1;
    }
    return newNode.far;
}
}

```

day30 (圆圈中最后剩下的数字)

面试题62. 圆圈中最后剩下的数字

难度 简单  58     

0,1,,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1：

输入：n = 5, m = 3
输出：3

示例 2：

输入：n = 10, m = 17
输出：2

限制：

- $1 \leq n \leq 10^5$
- $1 \leq m \leq 10^6$

```

class Solution {
    public int lastRemaining(int n, int m) {
        List<Integer> list = new ArrayList<>();
        for(int i=0;i<n;i++){
            list.add(i);
        }
    }
}

```

```

    }
    int start = 0;
    while(list.size()>1){
        int size = list.size();
        int delIndex = (start+m-1)%size;
        list.remove(delIndex);
        start = delIndex;
    }
    return list.get(0);
}
}

```

day31 (数组升序--冒泡、选择、插入、快速、归并、堆排序算法)

912. 排序数组

难度 中等  96     

给你一个整数数组 `nums`，请你将该数组升序排列。

示例 1：

输入：nums = [5,2,3,1]
输出：[1,2,3,5]

示例 2：

输入：nums = [5,1,1,2,0,0]
输出：[0,0,1,1,2,5]

提示：

1. `1 <= nums.length <= 50000`
2. `-50000 <= nums[i] <= 50000`

```

class Solution {
    public int[] sortArray(int[] nums) {
        // BubbleSort(nums);
        // selectSort(nums);
        insertSort(nums);
        return nums;
    }

    //冒泡排序
    public void BubbleSort(int[] nums){
        int n = nums.length;
        for(int i=0;i<n-1;i++){
            boolean isSort = true;
            for(int j=0;j<n-i-1;j++){
                if(nums[j+1]<nums[j]){
                    isSort = false;
                }
            }
            if(isSort){
                return;
            }
        }
    }
}

```

```

        int temp = nums[j+1];
        nums[j+1] = nums[j];
        nums[j] = temp;
    }
}
if(isSort){
    break;
}
}
}

//选择排序
public void selectSort(int[] nums){
    int n = nums.length;
    //遍历n-1次 前n-1选择好了之后 最后一个元素必然满足要求
    for(int i=0;i<n-1;i++){
        int minIndex = i;
        for(int j=i+1;j<n;j++){
            if(nums[j]<nums[minIndex]){
                minIndex = j;
            }
        }
        //交换数据把最小的元素给i
        int temp = nums[minIndex];
        nums[minIndex] = nums[i];
        nums[i] = temp;
    }
}

//插入排序
public void insertSort(int[] nums){
    int n = nums.length;
    //循环数组 将nums[i]插入到 nums[0,i)有序区间中
    for(int i=1;i<n;i++){
        int temp = nums[i];
        int j = i;
        while(j>0 && (nums[j-1]>temp)){
            nums[j] = nums[j-1];
            j--;
        }
        nums[j] = temp;
    }
}

//快速排序 ---重点掌握
public void quickSort(int[] nums,int low,int high){
    int i,j,base;
    if(low>high){
        return;
    }
    i = low;
    j = high;
    base = nums[low];
    while(i<j){
        //从右边找一个比base小的
        while(base<=nums[j] && (i<j)){
            j--;
        }
    }
}

```



```

        //从左边找一个比base大的
        while (base>=nums[i] && (i<j)){
            i++;
        }
        //交换i, j数据
        if(i<j){
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
    //交换base和ij相遇数字
    int temp = nums[i];
    nums[i] = base;
    nums[low] = temp; //不能写成base=temp i,j,base都只是 low high nums[low]的
    代表

    //递归执行base左边和右边
    quickSort(nums,low,j-1);
    quickSort(nums,j+1,high);
}
}

```

堆排序

```

import java.util.Arrays;

/**
 * Created by chengxiao on 2016/12/17.
 * 堆排序demo
 */
public class HeapSort {
    public static void main(String []args){
        int []arr = {9,8,7,6,5,4,3,2,1};
        sort(arr);
        System.out.println(Arrays.toString(arr));
    }
    public static void sort(int []arr){
        //1. 构建大顶堆
        for(int i=arr.length/2-1;i>=0;i--){
            //从第一个非叶子结点从下至上，从右至左调整结构
            adjustHeap(arr,i,arr.length);
        }
        //2. 调整堆结构+交换堆顶元素与末尾元素
        for(int j=arr.length-1;j>0;j--){
            swap(arr,0,j); //将堆顶元素与末尾元素进行交换
            adjustHeap(arr,0,j); //重新对堆进行调整
        }
    }

    /**
     * 调整大顶堆（仅是调整过程，建立在大顶堆已构建的基础上）
     * @param arr
     * @param i
     * @param length
     */
    public static void adjustHeap(int []arr,int i,int length){

```

```

        int temp = arr[i]; //先取出当前元素i
        for(int k=i*2+1; k<length; k=k*2+1){ //从i结点的左子结点开始，也就是2i+1处开始
            if(k+1<length && arr[k]<arr[k+1]){ //如果左子结点小于右子结点，k指向右子结
点
                k++;
            }
            if(arr[k] > temp){ //如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）
                arr[i] = arr[k];
                i = k;
            }else{
                break;
            }
        }
        arr[i] = temp; //将temp值放到最终的位置
    }

    /**
     * 交换元素
     * @param arr
     * @param a
     * @param b
     */
    public static void swap(int []arr, int a, int b){
        int temp=arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
    }
}

```

归并排序

```

public class MergeSort{
    2
    3
    4     public int[] sort(int[] sourceArray) throws Exception {
    5         // 对 arr 进行拷贝，不改变参数内容
    6         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
    7
    8         if (arr.length < 2) {
    9             return arr;
    10        }
    11        int middle = (int) Math.floor(arr.length / 2);
    12
    13        int[] left = Arrays.copyOfRange(arr, 0, middle);
    14        int[] right = Arrays.copyOfRange(arr, middle, arr.length);
    15
    16        return merge(sort(left), sort(right));
    17    }
    18
    19    protected int[] merge(int[] left, int[] right) {
    20        int[] result = new int[left.length + right.length];
    21        int i = 0;
    22        while (left.length > 0 && right.length > 0) {
    23            if (left[0] <= right[0]) {
    24                result[i++] = left[0];
    25                left = Arrays.copyOfRange(left, 1, left.length);
    26            } else {

```

```

27         result[i++] = right[0];
28         right = Arrays.copyOfRange(right, 1, right.length);
29     }
30 }
31
32 while (left.length > 0) {
33     result[i++] = left[0];
34     left = Arrays.copyOfRange(left, 1, left.length);
35 }
36
37 while (right.length > 0) {
38     result[i++] = right[0];
39     right = Arrays.copyOfRange(right, 1, right.length);
40 }
41
42 return result;
43 }
44
45 }

```

day32 (有效括号问题)

20. 有效的括号

难度 简单  1477     

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

输入: "()"

输出: true

示例 2:

输入: "()[]{}"

输出: true

示例 3:

输入: "()["

输出: false

```

class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

```

```

char[] chars = s.toCharArray();
for(char c:chars){
    if(stack.size()==0){
        stack.push(c);
    }else if(isPair(stack.peek(),c)){
        stack.pop();
    }else{
        stack.push(c);
    }
}
return stack.size()==0;
}

public boolean isPair(char a,char b){
    if(a=='(' && b==')' || a=='{' && b=='}' || a=='[' && b==']'){
        return true;
    }
    return false;
}
}

```

1111. 有效括号的嵌套深度

难度 中等  77     

有效括号字符串 定义：对于每个左括号，都能找到与之对应的右括号，反之亦然。详情参见题末「有效括号字符串」部分。

嵌套深度 depth 定义：即有效括号字符串嵌套的层数， $\text{depth}(A)$ 表示有效括号字符串 A 的嵌套深度。详情参见题末「嵌套深度」部分。

有效括号字符串类型与对应的嵌套深度计算方法如下图所示：

1. 空字符串	2. AB 型	3. (A) 型
""	$((()))()$	$((()))()$
$\text{depth} = 0$	$\text{depth} = \max(2, 1) = 2$	$\text{depth} = 1 + \text{depth}(A) = 3$

给你一个「有效括号字符串」 seq ，请你将其分成两个不相交的有效括号字符串， A 和 B ，并使这两个字符串的深度最小。

- 不相交：每个 $\text{seq}[i]$ 只能分给 A 和 B 二者中的一个，不能既属于 A 也属于 B 。
- A 或 B 中的元素在原字符串中可以不连续。
- $A.\text{length} + B.\text{length} = \text{seq}.\text{length}$
- 深度最小： $\max(\text{depth}(A), \text{depth}(B))$ 的可能取值最小。

划分方案用一个长度为 $\text{seq}.\text{length}$ 的答案数组 answer 表示，编码规则如下：

- $\text{answer}[i] = 0$ ， $\text{seq}[i]$ 分给 A 。
- $\text{answer}[i] = 1$ ， $\text{seq}[i]$ 分给 B 。

今天题目太难理解了，没意思

使 $\max(\text{depth}(A), \text{depth}(B))$ 的可能取值最小”。这句话其实相当于让A字符串和B字符串的depth尽可能的接近。为什么呢？因为seq对应的栈上，每个左括号都对应一个深度，而这个左括号，要么是A的，要么是B的。所以，栈上的左括号只要按奇偶分配给A和B就可以啦！时间复杂度很明显是 $O(n)O(n)$ 的，空间复杂度也是 $O(n)O(n)$ （如果算返回的变量的话）。

```
class Solution {
    public int[] maxDepthAfterSplit(String seq) {
        int[] ans = new int[seq.length()];

        for(int i=0;i<seq.length();i++){
            //奇数下标的( 分给A 偶数下标 ( 分给B
            //奇数下标的) 分给B 偶数下标) 分给A
            // if(seq.charAt(i)=='('){
            //     if(i%2==0){
            //         ans[i] = 1;
            //     }else{
            //         ans[i] = 0;
            //     }
            // }else{
            //     if(i%2==0){
            //         ans[i] = 0;
            //     }else{
            //         ans[i] = 1;
            //     }
            // }
            ans[i] = seq.charAt(i)=='('? i&1:(i+1)&1;
        }
        return ans;
    }
}
```

day33（生命游戏）

289. 生命游戏

难度 中等  156  收藏  分享  切换为英文  关注  反馈

根据 百度百科，生命游戏，简称为生命，是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机。

给定一个包含 $m \times n$ 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞都具有一个初始状态：1 即为活细胞（live），或 0 即为死细胞（dead）。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

1. 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；
2. 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；
3. 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；
4. 如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

根据当前状态，写一个函数来计算面板上所有细胞的下一个（一次更新后的）状态。下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。

示例：

输入：

```
[
  [0,1,0],
  [0,0,1],
  [1,1,1],
  [0,0,0]
]
```

输出：

```
[
  [0,0,0],
  [1,0,1],
  [0,1,1],
]
```

idea:

不能每个点周围判断完之后立即更新周围点的状态，因为当前点的周围状态会是下一次研究的中心点对象，采取标志位方式 2代表：alive-->dead -1: dead-->alive

```
class Solution {
    public int m;
    public int n;
    public void gameOfLife(int[][] board) {
        int[] dx = {0, 0, 1, -1, 1, 1, -1, -1};
        int[] dy = {1, -1, 0, 0, 1, -1, 1, -1};

        m = board.length;
        n = board[0].length;

        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                int cnt = 0; //统计当前细胞周围活细胞数量
                for(int k=0;k<8;k++){
                    int next_X = i+dx[k];
                    int next_Y = j+dy[k];
                    if(check(next_X,next_Y) && board[next_X][next_Y]>0){//标记为2
                        //或者-1只是一个标识位 在没有遍历完之前2, -1还代表他底层的1,0
                        cnt ++;
                    }
                }

                if(board[i][j]==1){
                    if(cnt<2 || cnt>3){
                        board[i][j] = 2; //活细胞-->死亡
                    }
                }else if(cnt==3){
                    board[i][j] = -1; //死细胞-->复活
                }
            }
        }

        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(board[i][j]==2){
                    board[i][j] = 0;
                }else if(board[i][j]==-1){
                    board[i][j] = 1;
                }
            }
        }

        public boolean check(int x,int y){
            if(x<0 || x>=m || y<0 || y>=n){
                return false;
            }
            return true;
        }
    }
}
```

day34 (字符串到整数)

8. 字符串转换整数 (atoi)

难度 中等

👍 635

♡ 收藏

🔗 分享

🌐 切换为英文

🔔 关注

🗉 反馈

请你来实现一个 `atoi` 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。接下来的转化规则如下：

- 如果第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字字符组合起来，形成一个有符号整数。
- 假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成一个整数。
- 该字符串在有效的整数部分之后也可能会存在多余的字符，那么这些字符可以被忽略，它们对函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换，即无法进行有效转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

提示：

- 本题中的空白字符只包括空格字符 ' '。
- 假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 $[-2^{31}, 2^{31} - 1]$ 。如果数值超过这个范围，请返回 `INT_MAX` ($2^{31} - 1$) 或 `INT_MIN` (-2^{31})。

示例 1：

输入："42"

输出：42

示例 2:

输入: " -42"

输出: -42

解释: 第一个非空白字符为 '-', 它是一个负号。

我们尽可能将负号与后面所有连续出现的数字组合起来, 最后得到 -42 。

示例 3:

输入: "4193 with words"

输出: 4193

解释: 转换截止于数字 '3' , 因为它的下一个字符不为数字。

示例 4:

输入: "words and 987"

输出: 0

解释: 第一个非空字符是 'w', 但它不是数字或正、负号。

因此无法执行有效的转换。

示例 5:

输入: "-91283472332"

输出: -2147483648

解释: 数字 "-91283472332" 超过 32 位有符号整数范围。

因此返回 INT_MIN (-2^{31}) 。

```
class Solution {
    public int myAtoi(String str) {
        char[] chars = str.toCharArray();
        int n = chars.length;
        int cur = 0;
        //去掉前导空格
        while(cur < n && chars[cur] == ' '){
            cur++;
        }
        if(cur == n){
            return 0;
        }

        boolean negative = false;
        if(chars[cur] == '-'){
            negative = true;
            cur++;
        } else if(chars[cur] == '+'){
            cur++;
        } else if(!Character.isDigit(chars[cur])){
            return 0;
        }

        int ans = 0;
        int digit = 0;
```



```

while(cur<n && Character.isDigit(chars[cur])){
    digit = chars[cur]-'0';
    if(ans>(Integer.MAX_VALUE-digit)/10){ //保证不溢出
        return ans = negative? Integer.MIN_VALUE:Integer.MAX_VALUE;
    }
    ans = ans*10+digit;
    cur++;
}
return negative?-ans:ans;
}
}

```

day35 (接雨水)

42. 接雨水

难度 困难 1117 收藏 分享 切换为英文 关注 反馈

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 $[0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]$ 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。感谢 Marcos 贡献此图。

示例：

输入： $[0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]$

输出：6

对于每一列来说，他能存的雨水量是他左边最高墙和右边最高墙中较低的那堵墙的高度减去自身墙的高度。所以可以用数组记录每列左右最高墙的高度，然后计算每一列可以存的雨水量

```

class Solution {
    public int trap(int[] height) {
        //*****暴力解法*****
        int ans = 0;
        //从第二个柱体遍历到倒数第二个柱体
        for(int i=1; i<height.length-1; i++){
            int leftMax = 0;
            int rightMax = 0;
            for(int j=0; j<=i; j++){
                leftMax = Math.max(leftMax, height[j]);
            }
            for(int k=i; k<height.length; k++){
                rightMax = Math.max(rightMax, height[k]);
            }
            ans += Math.min(leftMax, rightMax) - height[i];
        }
        return ans;

        //*****dp解法*****
    }
}

```

```

int n = height.length;
int ans = 0;
if(n==0){
    return 0;
}
int[][] dp = new int[n][2];
//dp[i][0] dp[i][1] 表示第i柱子左右两边的最大高度(包括当前柱子高度)
dp[0][0] = height[0]; //左
dp[n-1][1] = height[n-1]; //右
//填充柱子左边高度所有情况
for(int i=1;i<n;i++){
    dp[i][0] = Math.max(dp[i-1][0],height[i]);
}
for(int j=n-2;j>=0;j--){
    dp[j][1] = Math.max(dp[j+1][1],height[j]);
}
for(int k=0;k<n;k++){
    ans += Math.min(dp[k][0],dp[k][1])-height[k];
}
return ans;
}
}

```

day37 (编辑距离)

72. 编辑距离

难度 困难

👍 738

🤍 收藏

🔗 分享

🌐 切换为英文

🔔 关注

📝 反馈

给你两个单词 *word1* 和 *word2*，请你计算出将 *word1* 转换成 *word2* 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

示例 1：

```
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2：

```
输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
```

状态定义：

$dp[i][j]$ 表示 *word1* 的前 *i* 个字母转换成 *word2* 的前 *j* 个字母所使用的最少操作。

状态转移：

i 指向 *word1*，*j* 指向 *word2*

若当前字母相同，则 $dp[i][j] = dp[i - 1][j - 1]$ ；

否则取增删替三个操作的最小值 + 1，即：

$$dp[i][j] = \min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]) + 1$$

```
class Solution {
    public int minDistance(String word1, String word2) {
        int len1 = word1.length();
        int len2 = word2.length();
        //dp[i][j] word1的前i个字符转化为word2前j个字符所使用的最小操作次数
        int[][] dp = new int[len1+1][len2+1];
        for(int i = 0; i <= len1; i++){
            dp[i][0] = i;
        }
        for(int j = 0; j <= len2; j++){
```

```

        dp[0][j] = j;
    }

    for(int i=1;i<=len1;i++){
        for(int j=1;j<=len2;j++){
            if(word1.charAt(i-1) == word2.charAt(j-1)){
                dp[i][j] = dp[i-1][j-1];
            }else{
                dp[i][j] = Math.min(Math.min(dp[i][j-1],dp[i-1][j]),dp[i-1][j-1])+1;
            }
        }
    }
    return dp[len1][len2];
}
}

```

day38（原地旋转数组）

面试题 01.07. 旋转矩阵

难度 中等

👍 55

📖 收藏

🔗 分享

🌐 切换为英文

🔔 关注

📝 反馈

给你一幅由 $N \times N$ 矩阵表示的图像，其中每个像素的大小为 4 字节。请你设计一种算法，将图像旋转 90 度。

不占用额外内存空间能否做到？

示例 1:

给定 `matrix =`

```

[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],

```

原地旋转输入矩阵，使其变为：

```

[
  [7,4,1],
  [8,5,2],
  [9,6,3]
]

```

idea:

- 1、以对角线为轴进行旋转
- 2、每行以中点进行旋转

```

class Solution {
    public void rotate(int[][] matrix) {
        int m = matrix.length;
        //以对角线为轴进行翻转
        for(int i = 0;i<m;i++){
            for(int j = 0;j<m;j++){
                if(i<j){

```

```

        int temp = 0;
        temp = matrix[i][j];
        matrix[i][j] = matrix[j][i];
        matrix[j][i] = temp;
    }
}
}
//对每行数据以中心为轴进行进行翻转
int mid = m>>1;
for(int i=0;i<m;i++){
    for(int j=0;j<mid;j++){
        int temp = matrix[i][j];
        matrix[i][j] = matrix[i][m-1-j];
        matrix[i][m-1-j] = temp;
    }
}
}
}

```

day39 (机器人的运动范围)

面试题13. 机器人的运动范围

难度 中等

👍 78

📖 收藏

🔗 分享

🌐 切换为英文

🔔 关注

💡 反馈

地上有一个m行n列的方格，从坐标 $[0,0]$ 到坐标 $[m-1,n-1]$ 。一个机器人从坐标 $[0,0]$ 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 $[35,37]$ ，因为 $3+5+3+7=18$ 。但它不能进入方格 $[35,38]$ ，因为 $3+5+3+8=19$ 。请问该机器人能够到达多少个格子？

示例 1:

输入: $m = 2, n = 3, k = 1$
输出: 3

示例 1:

输入: $m = 3, n = 1, k = 0$
输出: 1

提示:

- $1 \leq n, m \leq 100$
- $0 \leq k \leq 20$

DFS:

```

class Solution {
    boolean[][] isvisited;
    public int movingCount(int m, int n, int k) {
        isvisited = new boolean[m][n];
        return dfs(0,0,m,n,k);
    }
    public int dfs(int x,int y,int m,int n,int k){
        if(x<0 || y<0 || x>=m || y>=n || isvisited[x][y]==true ||
        (x%10+x/10+y%10+y/10)>k){

```

```

        return 0;
    }
    isvisited[x][y] = true;
    return 1+dfs(x-1,y,m,n,k)+dfs(x+1,y,m,n,k)+dfs(x,y-1,m,n,k)+dfs(x,y+1,m,n,k);
}
}

```

BFS待续

day40 (括号生成--递归DFS)

22. 括号生成

难度 中等

👍 902

📖 收藏

🔗 分享

🌐 切换为英文

🔔 关注

📝 反馈

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。

示例：

输入：n = 3

输出：

```

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

```

```

class Solution {
    List<String> ans = new ArrayList<>();
    public List<String> generateParenthesis(int n) {
        dfs(n,n,"");
        return ans;
    }
    public void dfs(int left,int right,String curStr){
        if(left==0 && right==0){
            ans.add(curStr);
            return;
        }
        if(left>0){
            dfs(left-1,right,curStr+"(");
        }
        if(right>left){
            dfs(left,right-1,curStr+")");
        }
    }
}

```

day41 (反转字符串里的单词)

151. 翻转字符串里的单词

难度 中等

140

收藏

分享

切换为英文

关注

反馈

给定一个字符串，逐个翻转字符串中的每个单词。

示例 1:

输入: "the sky is blue"

输出: "blue is sky the"

示例 2:

输入: " hello world! "

输出: "world! hello"

解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。

示例 3:

输入: "a good example"

输出: "example good a"

解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

```
class Solution {
    public String reverseWords(String s) {
        String ss = s.trim();
        String[] SArrays = ss.split("\\s+");
        StringBuilder sb = new StringBuilder();
        for(int i=SArrays.length-1;i>0;i--){
            sb.append(SArrays[i]).append(" ");
        }
        sb.append(SArrays[0]);
        return sb.toString();
    }
}
```

day42（合并K个有序链表---优先队列（堆））

23. 合并K个排序链表

难度 困难

572

收藏

分享

切换为英文

关注

反馈

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

输入:

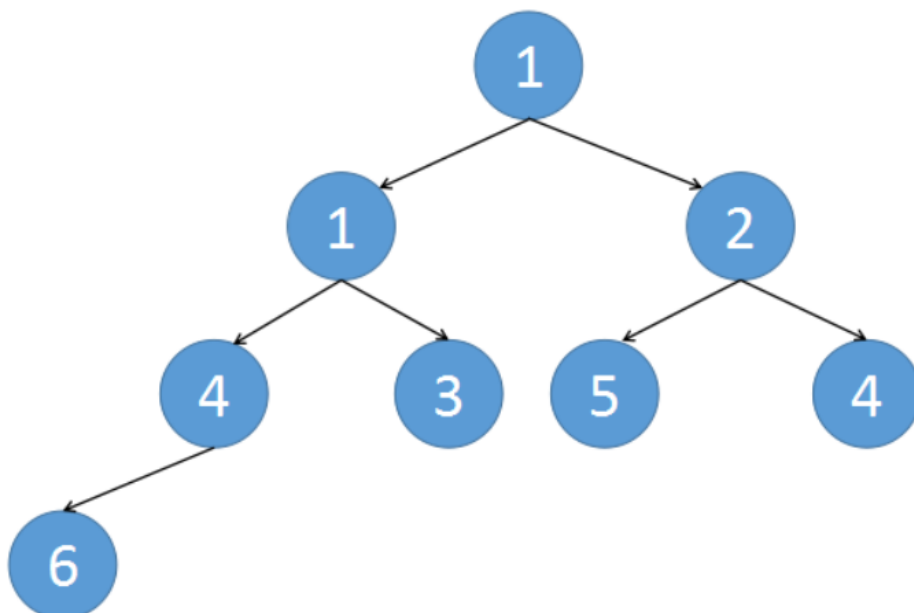
```
[
  1->4->5,
  1->3->4,
  2->6
]
```

输出: 1->1->2->3->4->4->5->6

这时候我们需要一种辅助数据结构-堆，有了堆这个数据结构，难度等级是困难的题目，瞬间变成简单了。我们把三个链表一股脑的全放到堆里面

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

然后由堆根据节点的 val 自动排好序



```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists==null || lists.length==0){
            return null;
        }
    }
}
```



```

    }
    int k = lists.length;
    PriorityQueue<ListNode> queue = new PriorityQueue<>(new
Comparator<ListNode>(){
    public int compare(ListNode o1,ListNode o2){
        //升序排序
        return (o1.val-o2.val);
    }
});


//把每个链表的数据放入queue
for(int i=0;i<k;i++){
    ListNode head = lists[i];
    while(head!=null){
        queue.add(head);
        head = head.next;
    }
}

ListNode temp = new ListNode(-1);
ListNode head = temp;
while(!queue.isEmpty()){
    temp.next = queue.poll();
    temp = temp.next;
}
temp.next = null;
return head.next;
}
}

```

day43（两个链表数据相加--stack）

445. 两数相加 II

难度 中等  177  收藏  分享  切换为英文  关注  反馈

给你两个 非空 链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶：

如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。

示例：

输入：(7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 8 -> 0 -> 7

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;

```

```

    *    ListNode(int x) { val = x; }
    * }
    */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        Stack<Integer> stack1 = new Stack<>();
        Stack<Integer> stack2 = new Stack<>();
        while(l1!=null){
            stack1.add(l1.val);
            l1 = l1.next;
        }



        while(l2!=null){
            stack2.add(l2.val);
            l2 = l2.next;
        }

        int carry = 0;
        ListNode head = null; //从尾部一个个插入链表
        while(!stack1.isEmpty() || !stack2.isEmpty() || carry>0){
            int sum = carry;
            sum += stack1.isEmpty() ? 0:stack1.pop();
            sum += stack2.isEmpty() ? 0:stack2.pop();
            ListNode node = new ListNode(sum%10);
            node.next = head;
            head = node;
            carry = sum/10;
        }
        return head;
    }
}

```

day44 (01矩阵--BFS VS 最远海洋问题)

542. 01 矩阵

难度 中等  229  收藏  分享  切换为英文  关注  反馈

给定一个由 0 和 1 组成的矩阵，找出每个元素到最近的 0 的距离。

两个相邻元素间的距离为 1 。

示例 1:

输入:

```

0 0 0
0 1 0
0 0 0

```

输出:

```

0 0 0
0 1 0
0 0 0

```

idea:

首先把每个源点 00 入队，然后从各个 00 同时开始一圈一圈的向 11 扩散（每个 11 都是被离它最近的 00 扩散到的），扩散的时候可以设置 `int[][] dist` 来记录距离（即扩散的层次）并同时标志是否访问过。对于本题是可以直接修改原数组 `int[][] matrix` 来记录距离和标志是否访问的，这里要注意先把 `matrix` 数组中 1 的位置设置成 -1（设成 `Integer.MAX_VALUE` 啦，`m * n` 啦，10000 啦都行，只要是个无效的距离值来标志这个位置的 1 没有被访问过就行辣~）

```
class Solution {
    public int[][] updateMatrix(int[][] matrix) {
        Queue<int[]> queue = new ArrayDeque<>();
        int m = matrix.length;
        int n = matrix[0].length;
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(matrix[i][j]==0){
                    queue.add(new int[] {i,j});
                }else{
                    matrix[i][j] = -1;
                }
            }
        }

        int[] dx = {-1,1,0,0};
        int[] dy = {0,0,-1,1};

        while(!queue.isEmpty()){
            int[] temp = queue.poll();
            int x = temp[0];
            int y = temp[1];
            for(int k =0;k<4;k++){
                int newX = x + dx[k];
                int newY = y + dy[k];
                if(newX>=0 && newX<m && newY>=0 && newY<n && matrix[newX]
[newY]==-1){
                    matrix[newX][newY] = matrix[x][y] + 1;
                    queue.add(new int[] {newX,newY});
                }
            }
        }
        return matrix;
    }
}
```

day45（合并区间）

56. 合并区间

难度 中等

👍 391

📖 收藏

🔗 分享

🌐 切换为英文

🔔 关注

📝 反馈

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

输入: `[[1,3],[2,6],[8,10],[15,18]]`

输出: `[[1,6],[8,10],[15,18]]`

解释: 区间 `[1,3]` 和 `[2,6]` 重叠，将它们合并为 `[1,6]`。

示例 2:

输入: `[[1,4],[4,5]]`

输出: `[[1,5]]`

解释: 区间 `[1,4]` 和 `[4,5]` 可被视为重叠区间。

idea:

一、合并 2 个区间

2 个区间的关系有以下 6 种，但是其实可以变成上面 3 种情况（只需要假设 第一个区间的起始位置 \leq 第二个区间的起始位置，如果不满足这个假设，交换这两个区间）。这 3 种情况的合并的逻辑都很好写。



```
class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, new Comparator<int[]>() {
            public int compare(int[] o1, int[] o2) {
                return o1[0] - o2[0];
            }
        });

        int[][] ans = new int[intervals.length][2];
        int count = -1;
        for (int[] interval : intervals) {
            if (count == -1 || interval[0] > ans[count][1]) {
                ans[++count] = interval;
            } else {
                ans[count][1] = Math.max(ans[count][1], interval[1]);
            }
        }
        return Arrays.copyOf(ans, count + 1);
    }
}
```

```
}
```

day46 (跳跃数组)

55. 跳跃游戏

难度 中等

👍 618

♡ 收藏

🔗 分享

🌐 切换为英文

🔔 关注

📝 反馈

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

输入: [2,3,1,1,4]

输出: true

解释: 我们可以先跳 1 步，从位置 0 到达 位置 1，然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

输入: [3,2,1,0,4]

输出: false

解释: 无论如何，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0，所以你永远不可能到达最后一个位置。

```
class Solution {
    public boolean canJump(int[] nums) {
        int max = 0;
        for(int i = 0; i < nums.length; i++){
            if(i > max){
                return false;
            }
            max = Math.max(max, i + nums[i]); // i + nums[i] 位置i所能达到的最远距离
        }
        return max >= nums.length - 1;
    }
}
```

day47 (盛最多水的容器)

11. 盛最多水的容器

难度 中等

1351

收藏

分享

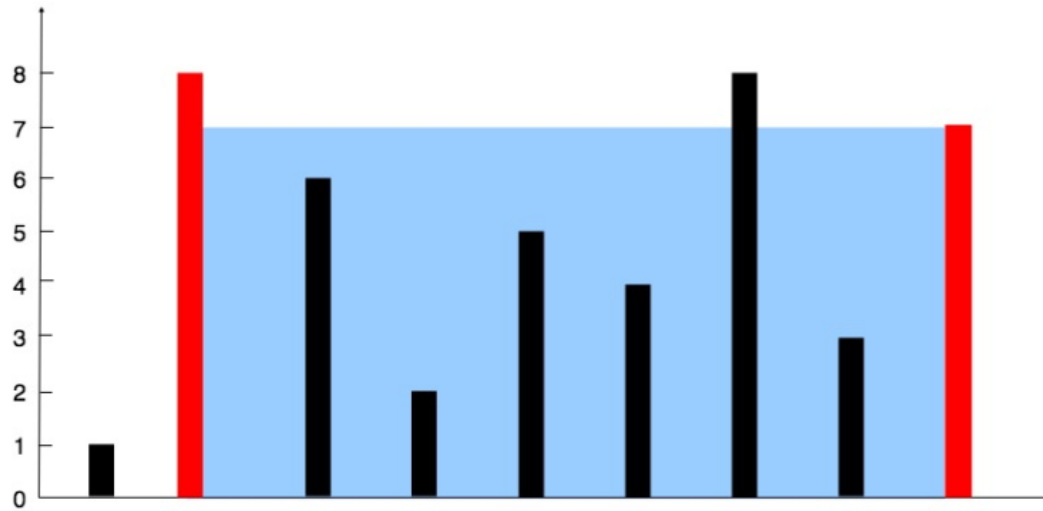
切换为英文

关注

反馈

给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 $[1, 8, 6, 2, 5, 4, 8, 3, 7]$ 。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

```
class Solution {
    public int maxArea(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }
        int left = 0;
        int right = height.length - 1;
        int maxArea = 0;
        int area = 0;
        while (left < right) {
            if (height[left] > height[right]) {
                area = height[right] * (right - left);
                right--;
            } else if (height[left] < height[right]) {
                area = height[left] * (right - left);
                left++;
            } else {
                area = height[left] * (right - left);
                right--;
                left++;
            }
            maxArea = Math.max(area, maxArea);
        }
        return maxArea;
    }
}
```

day48 (岛屿数量BFS)

200. 岛屿数量

难度 中等

👍 516

💖 收藏

🔗 分享

🌐 切换为英文

🔔 关注

🗉 反馈

给你一个由 '1'（陆地）和 '0'（水）组成的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1:

```
输入：
11110
11010
11000
00000
输出：1
```

示例 2:

```
输入：
11000
11000
00100
00011
输出：3
解释：每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。
```

```
class Solution {

    private class Node{
        int x = 0;
        int y = 0;
        public Node(int x,int y){
            this.x = x;
            this.y = y;
        }
    }

    public int m = 0;
    public int n = 0;
    public int[] dx = {-1,1,0,0};
    public int[] dy = {0,0,-1,1};
    public int numIslands(char[][] grid) {
        if(grid==null || grid.length==0){
            return 0;
        }

        m = grid.length;
        n = grid[0].length;
        int ans = 0;

        for(int i = 0;i<m;i++){
            for(int j = 0;j<n;j++){
```

```

        if(check(i,j) && grid[i][j]=='1'){
            ans++;
            bfs(grid,i,j);
        }
    }
}

return ans;

}

public boolean check(int x,int y){
    if(x<0 || x>=m || y<0 || y>=n){
        return false;
    }
    return true;
}

public void bfs(char[][] grid,int x,int y){
    Queue<Node> queue = new LinkedList<>();
    queue.add(new Node(x,y));
    grid[x][y] = 0;
    while(!queue.isEmpty()){
        Node curNode = queue.poll();
        for(int k = 0;k<4;k++){
            int next_x = curNode.x + dx[k];
            int next_y = curNode.y + dy[k];
            if(check(next_x,next_y) && grid[next_x][next_y]=='1'){
                queue.add(new Node(next_x,next_y));
                grid[next_x][next_y] = '0';
            }
        }
    }
}

}

}

```

day49 (优美子数组---滑动窗口)

1248. 统计「优美子数组」

难度 中等

👍 80

❤ 收藏

🔗 分享

🌐 切换为英文

🔔 关注

💬 反馈

给你一个整数数组 `nums` 和一个整数 `k`。

如果某个连续子数组中恰好有 `k` 个奇数数字，我们就认为这个子数组是「优美子数组」。

请返回这个数组中「优美子数组」的数目。

示例 1：

输入：nums = [1,1,2,1,1], k = 3

输出：2

解释：包含 3 个奇数的子数组是 [1,1,2,1] 和 [1,2,1,1]。

示例 2：

输入：nums = [2,4,6], k = 1

输出：0

解释：数列中不包含任何奇数，所以不存在优美子数组。

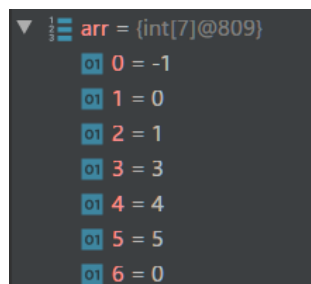
示例 3：

输入：nums = [2,2,2,1,2,2,1,2,2,2], k = 2

输出：16

例子：

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



```
class Solution {
    public int numberOfSubarrays(int[] nums, int k) {
        int len = nums.length;
        int res = 0;
        int oddCount = 0;
        int arr[] = new int[len + 2];
        //记录第oddCount个奇数的下标
        for (int i = 0; i < len; i++) {
            if ((nums[i] & 1) == 1) {
                arr[++oddCount] = i; //第++oddCount个奇数的下标是i
            }
        }
        arr[0] = -1; //左边界
        arr[oddCount + 1] = len; //右边界
    }
}
```




```

// arr[i]是窗口左边界
// arr[i+k-1] 是窗口右边界
// arr[i-1]是左边的上一个奇数，在此之后到arr[i]都可选
// arr[i+k]是右边的下一个奇数，在此之前都arr[i+k-1]都可选
//前面可选部分长度为arr[i]-arr[i-1]
//后面可选部分长度为arr[i+k]-arr[i+k-1]
//总的可能数等于前后可选的组合
//i+k<oddCount+2    i+k最大等于oddCount+1    arr[i+k] - arr[i+k-1] --
>arr[oddCount]-arr[oddCount-1] 边界情况
    for (int i = 1; i + k < oddCount + 2; i++) {
        res += (arr[i] - arr[i - 1]) * (arr[i + k] - arr[i + k - 1]);
    }
    return res;
}
}

```

day50 (二叉树的右视图)

199. 二叉树的右视图

难度 **中等**  217  收藏  分享  切换为英文  关注  反馈

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

输入: [1,2,3,null,5,null,4]

输出: [1, 3, 4]

解释:

```

      1             <---
     / \
    2   3           <---
     \   \
     5   4           <---

```

BFS层次遍历

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null){
            return res;
        }
    }
}

```

```

Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
while(!queue.isEmpty()){
    int size = queue.size();
    for(int i=0;i<size;i++){
        TreeNode node = queue.poll();
        if(node.left!=null){
            queue.offer(node.left);
        }
        if(node.right!=null){
            queue.offer(node.right);
        }
        if(i==size-1){
            res.add(node.val);
        }
    }
}
return res;
}
}

```

day51 (硬币组合)

面试题 08.11. 硬币

难度 中等  109     

硬币。给定数量不限的硬币，币值为25分、10分、5分和1分，编写代码计算n分有几种表示法。(结果可能会很大，你需要将结果模上1000000007)

示例1:

输入: n = 5

输出: 2

解释: 有两种方式可以凑成总金额:

5=5

5=1+1+1+1+1

示例2:

输入: n = 10

输出: 4

解释: 有四种方式可以凑成总金额:

10=10

10=5+5

10=5+1+1+1+1+1

10=1+1+1+1+1+1+1+1+1+1





```

class Solution {
    public int waysToChange(int n) {
        int[] coins = {25,10,5,1};
        int[] ans = new int[n+1];
        ans[0] = 1;
        for(int coin : coins){
            for(int i=coin;i<=n;i++){
                ans[i] = (ans[i]+ans[i-coin])%1000000007;
            }
        }
        return ans[n];
    }
}

```

day52 (全排列--BFS)

46. 全排列

难度 **中等**  665  收藏  分享  切换为英文  关注  反馈

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

示例:

输入: [1,2,3]

输出:

```

[
  [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) {
        List<List<Integer>> ans = new LinkedList<>();
        Queue<List<Integer>> queue = new LinkedList<>();
        queue.add(new LinkedList<Integer>());

        while(!queue.isEmpty()){
            List<Integer> list = queue.poll();
            int size = list.size();
            if(size == nums.length){
                ans.add(list);
                continue;
            }
            for(int i = 0; i<=nums.length-1;i++){
                if(!list.contains(nums[i])){
                    List<Integer> temp = new LinkedList<>(list);
                    temp.add(nums[i]);
                    queue.add(temp);
                }
            }
        }
    }
}

```

```
    }  
  }  
  return ans;  
}  
}
```

day53 (搜索旋转排序数组--二分法查找)

33. 搜索旋转排序数组

难度 中等  677     

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 `-1` 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

示例 1:

输入: `nums = [4,5,6,7,0,1,2]`, `target = 0`
输出: 4

示例 2:

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

idea:

33. 搜索旋转排序数组

题目要求时间复杂度 $O(\log n)$ ，显然应该使用二分查找。二分查找的过程就是不断收缩左右边界，而怎么缩小区间是关键。

如果数组「未旋转」，在数组中查找一个特定元素 `target` 的过程为：

- 若 `target == nums[mid]`，直接返回
- 若 `target < nums[mid]`，则 `target` 位于左侧区间 `[left, mid)` 中。令 `right = mid - 1`，在左侧区间查找
- 若 `target > nums[mid]`，则 `target` 位于右侧区间 `(mid, right]` 中。令 `left = mid + 1`，在右侧区间查找

但是这道题，由于数组「被旋转」，所以左侧或者右侧区间不一定是连续的。在这种情况下，如何判断 `target` 位于哪个区间？

根据旋转数组的特性，当元素不重复时，如果 `nums[i] <= nums[j]`，说明区间 `[i, j]` 是「连续递增」的。

`i`、`j` 可以重合，所以这里使用的比较运算符是「小于等于」

因此，在旋转排序数组中查找一个特定元素时：

- 若 `target == nums[mid]`，直接返回
- 若 `nums[left] <= nums[mid]`，说明左侧区间 `[left, mid]` 「连续递增」。此时：
 - 若 `nums[left] <= target <= nums[mid]`，说明 `target` 位于左侧。令 `right = mid - 1`，在左侧区间查找
 - 否则，令 `left = mid + 1`，在右侧区间查找
- 否则，说明右侧区间 `[mid, right]` 「连续递增」。此时：
 - 若 `nums[mid] <= target <= nums[right]`，说明 `target` 位于右侧区间。令 `left = mid + 1`，在右侧区间查找
 - 否则，令 `right = mid - 1`，在左侧区间查找
- 注意：区间收缩时不包含 `mid`，也就是说，实际收缩后的区间是 `[left, mid)` 或者 `(mid, right]`

```
class solution {
    public int search(int[] nums, int target) {
        if(nums==null || nums.length==0){
            return -1;
        }
        int left = 0;
        int right = nums.length-1;
        int mid = 0;
        while(left<=right){
            mid = (left+right)/2;
            if(nums[mid] == target){
                return mid;
            }
            if(nums[left]<=nums[mid]){
                if(target>=nums[left] && target<=nums[mid]){
                    right = mid-1;
                }else{
                    left = mid + 1;
                }
            }else{
                if(target>=nums[mid] && target<=nums[right]){
                    left = mid + 1;
                }else{
                    right = mid - 1;
                }
            }
        }
        return -1;
    }
}
```

day54 (数组中数字出现的次数)

面试题56 - I. 数组中数字出现的次数

难度 中等

133



一个整型数组 `nums` 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

示例 1:

输入: `nums = [4,1,4,6]`

输出: `[1,6]` 或 `[6,1]`

示例 2:

输入: `nums = [1,2,10,4,1,4,3,3]`

输出: `[2,10]` 或 `[10,2]`

限制:

- `2 <= nums <= 10000`

1、如果只有一个出现一次的数字，则结果为全部数字的异或和

2、两个出现一次的数字，需要

如果我们可以把所有数字分成两组，使得：

a:两个只出现一次的数字在不同的组中。

b:相同的数字会被分到相同的组中。

要点：选择所有数字异或和的bit位中，bit=1的二进制作为mask 对所有数字进行与操作，可以实现上面两个分组的要求。

```
class Solution {
    public int[] singleNumbers(int[] nums) {
        int sum = 0;
        for(int n:nums){
            sum ^= n;
        }
        //寻找两个不同数字的不同的bit位 即异或和bit=1
        int mask = 1;
        while((mask & sum) == 0){
            mask <<= 1;
        }
        int a = 0;
        int b = 0;
        for(int n:nums){
            if((mask & n) == 0){
                a ^= n;
            }
        }
    }
}
```

```

        }else{
            b ^= n;
        }
    }
    return new int[]{a,b};
}
}

```

day55 (无重复字符的最长子串----华为面试题目)

思路:

标签: 滑动窗口

定义一个 map 数据结构存储 (k, v), 其中 key 值为字符, value 值为字符位置 +1, 加 1 表示从字符位置后一个才开始不重复

我们定义不重复子串的开始位置为 start, 结束位置为 end

随着 end 不断遍历向后, 会遇到与 [start, end] 区间内字符相同的情况, 此时将字符作为 key 值, 获取其 value 值, 并更新 start, 此时 [start, end] 区间内不存在重复字符

无论是否更新 start, 都会更新其 map 数据结构和结果 ans。

```

public static int numofchars2(String s){
    int n = s.length();
    int ans = 0;
    Map<Character,Integer> map = new HashMap<>();
    for(int start = 0,end = 0; end<n; end++){
        char c = s.charAt(end);
        if(map.containsKey(c)){
            start = Math.max(map.get(c),start);
        }
        ans = Math.max(ans,end-start+1);
        map.put(s.charAt(end),end+1);
    }
    return ans;
}

```

day56 (重构二叉树)

面试题07. 重建二叉树

难度 中等

110



输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

例如，给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

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

递归解析：

递推参数： 前序遍历中根节点的索引pre_root、中序遍历左边界in_left、中序遍历右边界in_right。

终止条件： 当 in_left > in_right，子树中序遍历为空，说明已经越过叶子节点，此时返回 null。

递推工作：

建立根节点root： 值为前序遍历中索引为pre_root的节点值。

搜索根节点root在中序遍历的索引i： 为了提升搜索效率，本题解使用哈希表 dic 预存储中序遍历的值与索引的映射关系，每次搜索的时间复杂度为 O(1)。

构建根节点root的左子树和右子树： 通过调用 recur() 方法开启下一层递归。

左子树： 根节点索引为 pre_root + 1，中序遍历的左右边界分别为 in_left 和 i - 1。

右子树： 根节点索引为 i - in_left + pre_root + 1（即：根节点索引 + 左子树长度 + 1），中序遍历的左右边界分别为 i + 1 和 in_right。

返回值： 返回 root，含义是当前递归层级建立的根节点 root 为上一递归层级的根节点的左或右子节点。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    HashMap<Integer,Integer> map = new HashMap<>();
    int[] OriPreOrder;
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        OriPreOrder = preorder;
        for(int i=0;i<preorder.length;i++){
            map.put(inorder[i],i);
        }
        return recur(0,0,inorder.length-1);
    }
}
```

```

    }

    public TreeNode recur(int pre_root, int in_left, int in_right){
        if(in_left > in_right){
            return null;
        }
        TreeNode root = new TreeNode(OriPreOrder[pre_root]);
        int i = map.get(OriPreOrder[pre_root]);
        root.left = recur(pre_root+1, in_left, i-1);
        root.right = recur(pre_root+(i-in_left)+1, i+1, in_right);
        return root;
    }
}

```

day57 (Z型遍历二叉树)

```

class Node {
    char value; //数据域
    Node left; //左孩子节点
    Node right; //右孩子节点

    //}
    public Node(char value) {
        this.value = value;
    }
}

public ArrayList<ArrayList<Character>> zigzagOrder(Node root) {
    int level = 1; //指示当前遍历的层数
    Stack<Node> stack1 = new Stack<>(); //栈1存奇数节点
    stack1.push(root); //将根节点入栈
    Stack<Node> stack2 = new Stack<>(); //栈2存偶数节点
    ArrayList<ArrayList<Character>> list = new ArrayList<>();
    while (!stack1.empty() || !stack2.empty()) {
        if (level % 2 != 0) { //奇数层,该层为奇数层,叶子节点从右向左入栈,所以
            该层的叶子节点应入偶数栈
            ArrayList<Character> t = new ArrayList<>();
            while (!stack1.empty()) {
                Node cur = stack1.pop();
                if (cur != null) {
                    t.add(cur.value);
                    stack2.push(cur.left);
                    stack2.push(cur.right);
                }
            }
            if (!t.isEmpty()) {
                list.add(t);
                level++;
            }
        } else {
            ArrayList<Character> t = new ArrayList<>();
            while (!stack2.empty()) {
                Node cur = stack2.pop();
                if (cur != null) {
                    t.add(cur.value);
                    stack1.push(cur.right);
                    stack1.push(cur.left);
                }
            }
            if (!t.isEmpty()) {
                list.add(t);
                level++;
            }
        }
    }
    return list;
}

```

```

        }
    }
    if (!t.isEmpty()) {
        list.add(t);
        level++;
    }
}
}
return list;
}

```

day58 (对称二叉树)

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```

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

```

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

```

      1
     /\
    2  2
     \  \
     3   3

```

```

//递归
class solution {
    public boolean issymmetric(TreeNode root) {
        return check(root, root);
    }

    public boolean check(TreeNode p, TreeNode q) {
        if (p == null && q == null) {
            return true;
        }
        if (p == null || q == null) {
            return false;
        }
        return p.val == q.val && check(p.left, q.right) && check(p.right,
q.left);
    }
}

```

//迭代

```

class Solution {
    public boolean isSymmetric(TreeNode root) {
        return check(root, root);
    }

    public boolean check(TreeNode u, TreeNode v) {
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.offer(u);
        q.offer(v);
        while (!q.isEmpty()) {
            u = q.poll();
            v = q.poll();
            if (u == null && v == null) {
                continue;
            }
            if ((u == null || v == null) || (u.val != v.val)) {
                return false;
            }

            q.offer(u.left);
            q.offer(v.right);

            q.offer(u.right);
            q.offer(v.left);
        }
        return true;
    }
}

```

day59 (多少质数的和等于输入的这个整数)

给定一个正整数，编写程序计算有多少对质数的和等于输入的这个正整数，并输出结果。输入值小于1000。

如，输入为10, 程序应该输出结果为2。（共有两对质数的和为10,分别为(5,5),(3,7)）

输入描述:

输入包括一个整数n, ($3 \leq n < 1000$)

```

package ICBC;

import java.util.Scanner;

public class Test11 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()){
            int n = scanner.nextInt();
            int count = 0;
            for (int j=1;j<=n;j++){
                if(isPrime(j)){
                    System.out.print(j+" ");
                }
            }
        }
    }
}

```

```

        for(int i=2;i<=n/2;i++){
            if(isPrime(i) && isPrime(n-i)){
                count ++;
            }
        }
        System.out.println(count);
    }
}

public static boolean isPrime(int num){
    if(num<=1){
        return false;
    }
    if(num==2){
        return true;
    }

    boolean flag = true;
    for(int i =2;i<=Math.sqrt(num);i++){
        if(num%i==0){
            flag = false;
            break;
        }
    }
    if(flag){
        return true;
    }else{
        return false;
    }
}
}

```