剑指offer题目

15. 二级制中1的个数

包括正负数

- 无符号右移 >>>

```
/** 无符号右移 >>>
 * 执行耗时:1 ms,击败了97.57% 的Java用户
 * 内存消耗:35.2 MB,击败了83.96% 的Java用户
 */
public int hammingWeight(int n) {
    int res = 0;
    while(n != 0){
        res += (n&1);
        n >>>= 1;
    }
    return res;
}
```

- n&（n-1）

```
/**
 * 执行耗时:1 ms,击败了97.57% 的Java用户
 * 内存消耗:35.2 MB,击败了82.90% 的Java用户
 *
 */
public int hammingWeight(int n) {
    int res = 0;
    while(n != 0){
        n &= (n-1);
        res ++;
    }
    return res;
}
```

16.数值的整数次方

实现函数double Power(double base, int exponent)，求base的exponent次方

- 类似二分的思想

```
/**
 执行耗时:1 ms,击败了97.66% 的Java用户
 内存消耗:36.5 MB,击败了86.43% 的Java用户
 */
```

```
    public double myPow(double x, int n) {
        boolean positive = n > 0;
        double res = calculatePow(x,n);
        return positive? res : 1.0 /res;
    }

    private double calculatePow(double x,int n){
        if(n == 0)
           return 1;
        double y = calculatePow(x,n / 2);
        return n % 2 == 0 ? y  *y : y * y * x;

    }
```

- 快速幂
  - 比如x=2，n=10。n二进制表示就是1010，每个1就是x的对应次方数，相当于2^2*2^8，只需要找到每个1在二进制里面的位置，然后x对应次方就完了

```
/**
     * 快速幂
     执行耗时:1 ms,击败了97.66% 的Java用户
     内存消耗:37.8 MB,击败了36.93% 的Java用户
     */
    public double myPow(double x, int n) {
        double res = 1;
        boolean positive = n > 0;
        while(n != 0){
            if((n & 1) == 1){
                res *= x;
            }
            x *=x;
            n /= 2;
        }
        return positive ? res : 1.0 / res;
    }
```

18.删除链表的节点

- 我自己的做法

```
/**
     * 执行耗时:0 ms,击败了100.00% 的Java用户
     * 内存消耗:37.9 MB,击败了67.63% 的Java用户
     */
    public ListNode deleteNode(ListNode head, int val) {
        if(head.val == val)
            return head.next;
        ListNode pre = null;
        ListNode cur = head;
        while(cur != null){
            if(cur.val == val){
                pre.next = cur.next;
                break;
            }
```

```
            pre = cur;
            cur = cur.next;
        }
        return head;
    }
```

- 引入一个新的开头节点，便于后续统一的操作

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:37.8 MB,击败了75.11% 的Java用户
     */
    public ListNode deleteNode(ListNode head, int val) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode pre = dummy;
        ListNode cur = head;
        while(cur != null){
            if(cur.val == val){
                pre.next = cur.next;
                break;
            }
            pre = cur;
            cur = cur.next;
        }
        return dummy.next;
    }
```

leetcode原书是在O（1）的时间复杂度下删除给定的节点，思路：将后一个节点的内容复制到要删除的节点，要删除的节点指向后面第二个节点

- 关于删除链表重复节点的问题见原作122页

21. 调整数组顺序使奇数位于偶数前面

- 两个指针交换

```
/**
     执行耗时:2 ms,击败了98.94% 的Java用户
     内存消耗:46.4 MB,击败了63.05% 的Java用户
     */
    public int[] exchange(int[] nums) {
        int left = 0;
        int right = nums.length-1;
        while(left < right){
            while((nums[left] & 1) == 1 && left < right){
                left ++;
            }
            while((nums[right] & 1) == 0 && left < right){
                right --;
            }
            swap(nums,left,right);
        }
```

```
        return nums;
    }

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

22.链表中倒数第K个节点

- 快慢指针：让快指针先往前跑K个位置，然后快慢指针一起跑

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:36.5 MB,击败了40.92% 的Java用户
     */
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode slow = head;
        ListNode fast = head;
        while(k > 0){
            fast = fast.next;
            k--;
        }
        while(fast != null){
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
```

24.反转链表

- 熟练它

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:38.1 MB,击败了79.52% 的Java用户
     */
    public ListNode reverseList(ListNode head) {
        ListNode pre = null;
        ListNode cur = head;
        while(cur != null){
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
```

25.合并两个排序的链表

- 添加辅助头结点，两个指针分别指向两个链表

```
/**
     执行耗时:1 ms,击败了98.60% 的Java用户
     内存消耗:38.7 MB,击败了54.96% 的Java用户
     */
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode cur = dummy;
        while(l1 != null && l2 != null){
            if(l1.val < l2.val){
                cur.next = l1;
                l1 = l1.next;
            }
            else{
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }

        if(l1 != null){
            cur.next = l1;
        }
        else {
            cur.next = l2;
        }
        return dummy.next;
    }
```

26.树的子结构（这里要好好体会递归的想法）

- 分别判断某个节点开始能否满足条件

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:40.2 MB,击败了61.40% 的Java用户
     */
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(B == null || A == null)
            return false;
        // 当前节点能否满足条件 || 左子树满足条件 || 右子树满足条件
        return help(A,B) || isSubStructure(A.left,B) ||
isSubStructure(A.right,B);
    }

    // 判断某个节点开始能否满足条件
    private boolean help(TreeNode root,TreeNode target){
        if(target == null)
            return true;

        if(root == null || target.val != root.val)
```

```
            return false;

        return help(root.left,target.left) && help(root.right,target.right);
    }
```

27. 二叉树的镜像

- 按节点递归地往下走

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:35.7 MB,击败了78.77% 的Java用户
     */
    public TreeNode mirrorTree(TreeNode root) {
        if(root == null)
            return null;
        TreeNode temp = root.right;
        root.right = root.left;
        root.left = temp;
        mirrorTree(root.left);
        mirrorTree(root.right);
        return root;
    }
```

28.对称的二叉树（体会树递归的思想）

- 递归判断两个节点是否一样

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:36.2 MB,击败了90.77% 的Java用户
     */
    public boolean isSymmetric(TreeNode root) {
        if(root == null)
            return true;
        return help(root,root);
    }

    private boolean help(TreeNode left,TreeNode right){
        if(left == null && right == null)
            return true;
        if(left != null && right != null){
            if(left.val == right.val)
                return help(left.left,right.right) &&
help(left.right,right.left);
            return false;
        }
        return false;
    }
```

29.顺时针打印矩阵

- 注意判断边界的条件

```
/**
     执行耗时:1 ms,击败了97.22% 的Java用户
     内存消耗:39.6 MB,击败了89.59% 的Java用户
     */
    public int[] spiralOrder(int[][] matrix) {
        int row = matrix.length;
        if(row == 0)
            return new int[]{};
        int col = matrix[0].length;
        if(col == 0)
            return new int[]{};
        int [] res = new int[row * col];
        int index = 0;
        int left = 0,right = col - 1,top = 0,button = row-1;
        while(true){
            for(int i = left ; i <= right ; i++){
                res[index] = matrix[top][i];
                index ++;
            }
            top++;
            // 每次都要判断
            if(top > button)
                break;


            for(int i = top ; i <= button ; i++){
                res[index] = matrix[i][right];
                index ++;
            }
            right--;
            if(left > right)
                break;


            for (int i = right; i >= left; i--) {
                res[index] = matrix[button][i];
                index++;
            }
            button--;
            if(button < top)
                break;


            for(int i = button ; i >= top ; i--){
                res[index] = matrix[i][left];
                index ++;
            }
            left++;
            if(left > right)
                break;

        }
        return res;
```

```
    }
```

30.包含min函数的栈 push、pop、min时间复杂度都为O（1）

- 设置辅助栈，保持大小与数据栈一致，每次push时判断如果x小于辅助栈栈顶元素，则辅助栈`push`x，否则`push`辅助栈的栈顶元素

```
class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack();
        minStack = new Stack();
    }

    public void push(int x) {
        stack.push(x);
        if(minStack.isEmpty()){
            minStack.push(x);
        }
        else{
            // 精髓在这里 辅助栈的push规则
            int temp = minStack.peek();
            if(temp <= x)
                minStack.push(temp);
            else
                minStack.push(x);
        }
    }

    public void pop() {
        stack.pop();
        if(!minStack.isEmpty()){
            minStack.pop();
        }
    }

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

    public int min() {
        if(!minStack.isEmpty())
            return minStack.peek();
        return -1;
    }
}
```

31.栈的压入、弹出序列

- 思路：设置一个辅助栈，每次往栈中`push`一个数据，然后从弹出序列中往后匹配
- 注意：`java`更加建议使用`Deque stack = new LinkedList<>()`的方式来创建栈

```
/**
     执行耗时:2 ms,击败了94.61% 的Java用户
     内存消耗:38.3 MB,击败了41.37% 的Java用户
     */
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        Deque<Integer> stack = new LinkedList<>();
        int index = 0;
        for(int item : pushed){
            stack.push(item);
            while(!stack.isEmpty() && stack.peek() == popped[index]){
                stack.pop();
                index ++;
            }
        }
        return index == popped.length;
    }
```

32. 从上到下打印二叉树（二叉树层序遍历）

- 使用`Queue`实现`Queue queue = new LinkedList<>()`

```
/**
     执行耗时:1 ms,击败了99.75% 的Java用户
     内存消耗:38.4 MB,击败了85.66% 的Java用户
     */
    public int[] levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        if(root == null)
            return new int[0];
        List<Integer> res = new ArrayList<>();
        queue.offer(root);
        while(!queue.isEmpty()){
            int size = queue.size();
            for(int i = 0 ; i < size ; i++){
                TreeNode temp = queue.poll();
                res.add(temp.val);
                if(temp.left != null)
                    queue.offer(temp.left);
                if(temp.right != null)
                    queue.offer(temp.right);
            }
        }
        int[] array = new int[res.size()];
        for(int i = 0 ; i< array.length ; i++){
            array[i] = res.get(i);
        }
        return array;
    }
```

- 递归层次遍历（逐层打印）

```
class Solution {
    List<List<Integer>> node=new ArrayList();
    public List<List<Integer>> levelOrder(TreeNode root) {
        lei(root,0);
        return node;
    }
    public void lei(TreeNode root,int k){
        if(root!=null){
            if(node.size()<=k)
                node.add(new ArrayList());
            node.get(k).add(root.val);
            lei(root.left,k+1);
            lei(root.right,k+1);
        }
    }
}
```

32.之字形打印二叉树

- `List.add(0,val)`API使用

```
List<List<Integer>> res = new ArrayList<>();
        public List<List<Integer>> levelOrder(TreeNode root) {
            help(root,0);
            return res;
        }

        private void help(TreeNode root,int level){
            if(root == null)
                return;
            if(res.size() == level)
                res.add(new ArrayList<>());
            if((level & 1) == 0){
                res.get(level).add(root.val);
            }
            else{
                // 精髓所在
                res.get(level).add(0,root.val);
            }
            help(root.left,level+1);
            help(root.right,level+1);
        }
```

33.二叉搜索树的后序遍历序列

- 思路：以序列最后一个元素为根，可以将前面的序列分成两个部分
- 递归结束条件：1. `left == right` 只有一个节点　　2. `left > right` 没有节点了
- 先找到第一个比根大的元素，继续往右遍历，如果有比根小的，说明不能构成搜索树

```
/**
    执行耗时:0 ms,击败了100.00% 的Java用户
```

```
     内存消耗:35.8 MB,击败了85.65% 的Java用户
     */
    public boolean verifyPostorder(int[] postorder) {
        return help(postorder,0,postorder.length-1);
    }

    private boolean help(int[] postorder,int begin,int end){
        if(begin >= end)
            return true;
        int root = postorder[end];
        int i = begin;
        for(; i < end ; i++){
            if(postorder[i] > root)
                break;
        }

        // 继续往右遍历
        int j = i;
        for(; j < end ; j++){
            // 右子树有比根小的元素
            if(postorder[j] < root)
                return false;
        }
        // i的位置是第一个比根大的元素的索引
        return  help(postorder,begin,i - 1) && help(postorder, i , end - 1);
    }
```

34. 二叉树中和某一值的路径

- 体会回溯思想
- 注意`treeNode == null`与`treeNode.left == null && treeNode.right == null`的区别
- 到达叶节点为啥不`return`？`return`应该写在哪里?

```
/**
     执行耗时:1 ms,击败了100.00% 的Java用户
     内存消耗:38.5 MB,击败了95.24% 的Java用户
     */
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        dfs(new ArrayList<>(),root,0,sum);
        return res;
    }

    private void dfs(List<Integer> temp,TreeNode treeNode,int tempSum,int
sum){
        if(treeNode == null){
            return;
        }

        tempSum += treeNode.val;
        temp.add(treeNode.val);
        //到达叶节点
        if(treeNode.left == null && treeNode.right == null){
            if(tempSum == sum)
```

```
                res.add(new ArrayList<>(temp));
            return;
        }

        dfs(temp,treeNode.left,tempSum,sum);
        dfs(temp,treeNode.right,tempSum,sum);
        temp.remove(temp.size()-1);
    }
```

35.复杂链表的复制

- 使用map保存原始链表的复制

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:38.3 MB,击败了43.17% 的Java用户
     */
    public Node copyRandomList(Node head) {
        if(head == null)
            return null;
        Map<Node,Node> map = new HashMap<>();
        Node cur = head;
        while(cur != null){
            map.put(cur,new Node(cur.val));
            cur = cur.next;
        }

        cur = head;
        while(cur != null){
            // 精髓所在
            map.get(cur).next = map.get(cur.next);
            map.get(cur).random = map.get(cur.random);
            cur = cur.next;
        }
        return map.get(head);
    }
```

- 原地在每个节点后面加上一个新的复制

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:38.1 MB,击败了61.13% 的Java用户
     */
    // 原地修改
    public Node copyRandomList(Node head) {
        if(head == null)
            return null;
        Node cur = head;
        // 在每个节点的后面添加一个复制
        while(cur != null){
            Node temp = new Node(cur.val);
            temp.next = cur.next;
            cur.next = temp;
            cur = cur.next.next;
```

```
        }
        // 安排random指针
        cur = head;
        while(cur != null){
            if(cur.random != null){
                cur.next.random = cur.random.next;
            }
            cur = cur.next.next;
        }

        // 拆解成两个链表
        cur = head;
        Node res = head.next;
        Node curCopy = head.next;
        while(cur != null){
            cur.next = cur.next.next;
            cur = cur.next;
            if(curCopy.next != null){
                curCopy.next = curCopy.next.next;
                curCopy = curCopy.next;
            }
        }
        return res;
    }
```

36. 二叉搜索树与双向链表

- 中序遍历的模版

```
private void midOrder(Node node){
        if(node == null)
            return;
        midOrder(node.left);
        System.out.print(node);
        midOrder(node.right);
    }
```

- 中序遍历，再处理左右节点的引用

```
  /**
     执行用时：1 ms, 在所有 Java 提交中击败了18.74%的用户
     内存消耗：37.7 MB, 在所有 Java 提交中击败了83.32%的用户
     */
    private List<Node> list = new ArrayList<>();
    public Node treeToDoublyList(Node root) {
        if(root == null)
            return null;
        midOrder(root);
        for(int i = 0 ; i < list.size() ; i++){
            int left = i == 0 ? list.size() - 1 : i - 1;
            int right = i == list.size() - 1 ? 0 : i + 1;
```

```
            list.get(i).left = list.get(left);
            list.get(i).right = list.get(right);
        }
        return list.get(0);
    }

    private void midOrder(Node node){
        if(node != null) {
            midOrder(node.left);
            list.add(node);
            midOrder(node.right);
        }
    }
```

- 直接在中序遍历中修改引用，好好体会中序遍历的过程，理解节点的访问顺序

```
/**
     执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
     内存消耗：37.8 MB, 在所有 Java 提交中击败了63.26%的用户
     */
    private Node head,pre;
    public Node treeToDoublyList(Node root) {
        if(root == null)
            return null;
        midOrder(root);
        head.left = pre;
        pre.right = head;
        return head;
    }

    private void midOrder(Node node){
        if(node == null)
            return;
        midOrder(node.left);
        // 处理第一个节点
        if(pre == null)
            head = node;
        else{
            pre.right = node;
            node.left = pre;
        }
        pre = node;
        midOrder(node.right);
    }
```

38.字符串排列

- 回溯法解决
- 注意 `StringBuilder` 的API `sb.deleteCharAt(index)` 去掉某个位置的字符

```
        Set<String> set = new HashSet<>();
    public String[] permutation(String s) {
```

```
        help(s,new boolean[s.length()],new StringBuilder());
        String[] res = new String[set.size()];
        Iterator<String> ite = set.iterator();
        int index = 0;
        while(ite.hasNext()){
            res[index++] = ite.next();
        }
        return res;
    }

    private void help(String s,boolean[] visited,StringBuilder temp){
        if(temp.length() == s.length()){
            set.add(temp.toString());
            return;
        }
        for(int i = 0 ; i < s.length() ; i++){
            if(visited[i] == false){
                temp.append(s.charAt(i));
                visited[i] = true;
                help(s,visited,temp);
                visited[i] = false;
                temp.deleteCharAt(temp.length()-1);
            }
        }
    }
```

```
/**
     执行耗时:14 ms,击败了48.08% 的Java用户
     内存消耗:42.6 MB,击败了87.49% 的Java用户
     */
    public String[] permutation(String s) {
        char[] chars = s.toCharArray();
        Arrays.sort(chars);
        dfs(chars,new StringBuilder(),new boolean[s.length()]);
        // 这个API需要学习一下
        return res.toArray(new String[res.size()]);
    }
    List<String> res = new ArrayList<>();

    private void dfs(char[] s,StringBuilder temp,boolean[] visited){
        if(temp.length() == s.length){
            res.add(temp.toString());
            return;
        }
        for(int i = 0 ; i < s.length ; i++){
            if(visited[i])
                continue;
            if(i > 0 && s[i] == s[i - 1] && visited[i - 1])
                continue;
            temp.append(s[i]);
            visited[i] = true;
            dfs(s,temp,visited);
            // 注意这个API
            temp.deleteCharAt(temp.length()-1);
            visited[i] = false;
```

```
        }
    }
```

39.数组中出现次数超过一半的值

- 1. 排序求中间 2. `hashmap` 3. 摩尔投票（类似同归于尽）

```
/**
    执行耗时:1 ms,击败了99.99% 的Java用户
    内存消耗:41.5 MB,击败了85.76% 的Java用户
    */
  public int majorityElement(int[] nums) {
      int count = 0;
      int res = nums[0];
      for(int num : nums){
          if(count == 0)
              res = num;
          if(res == num)
              count++;
          else
              count--;
      }
      return res;
  }
```

41.数组的中位数

- 两个堆
- 最大堆`PriorityQueue<Integer> maxHeap = new PriorityQueue<>((o1,o2)->{return o2-o1;})`
- 最小堆`PriorityQueue<Integer> minHeap = new PriorityQueue<>()`

```
class MedianFinder {
    /**
     执行耗时:78 ms,击败了90.91% 的Java用户
     内存消耗:49.7 MB,击败了47.69% 的Java用户
     */
    private PriorityQueue<Integer> minHeap;
    private PriorityQueue<Integer> maxHeap;
    /** initialize your data structure here. */
    public MedianFinder() {
        minHeap = new PriorityQueue<>();
        maxHeap = new PriorityQueue<>((o1,o2)->{return o2-o1;});
    }

    public void addNum(int num) {
        if(num < findMedian()){
            maxHeap.offer(num);
```

```
        }
        else {
            minHeap.offer(num);
        }

        // 右边多
        if(maxHeap.size() < minHeap.size()){
            maxHeap.offer(minHeap.poll());
        }
        // 左边比右边多俩
        else if(maxHeap.size() - minHeap.size() > 1){
            minHeap.offer(maxHeap.poll());
        }
    }

    public double findMedian() {
        // 这个位置需要特别处理，第一次调用的时候要返回值
        if(maxHeap.size() == 0 && minHeap.size() == 0){
            return 0;
        }
        if(maxHeap.size() == minHeap.size())
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        else
            return maxHeap.peek();
    }
}
```

42.连续子数组的最大和

- 思路：从前往后求和，和为负数就重新设置为当前值

```
/**
     执行耗时:1 ms,击败了98.13% 的Java用户
     内存消耗:44.9 MB,击败了70.42% 的Java用户
     */
    public int maxSubArray(int[] nums) {
        int curSum = 0;
        // 这个结果的初始化有讲究的
        int max = Integer.MIN_VALUE;
        for(int i : nums){
            // 如果当前的和小于0,加上这个和会让求和变小，舍弃
            if(curSum < 0)
                curSum = i;
            else
                curSum += i;
            max = Math.max(max,curSum);
        }
        return max;
    }
```

- 动态规划：状态转移：`dp[i] = Math.max(dp[i-1] + nums[i],nums[i])  max = Math.max(max,dp[i])`
- 如果`dp[i-1]`是负数，加上`nums[i]`还不如`nums[i]`本身

```
/**
     执行耗时:1 ms,击败了98.13% 的Java用户
     内存消耗:44.6 MB,击败了96.66% 的Java用户
     */
    public int maxSubArray(int[] nums){
        int[] dp = new int[nums.length];
        int res = Integer.MIN_VALUE;
        for(int i = 0 ; i < nums.length ; i++){
            if(i > 0 && dp[i - 1] > 0){
                dp[i] = dp[i - 1] + nums[i];
            }
            else {
                dp[i] = nums[i];
            }
            res = Math.max(res,dp[i]);
        }
        return res;
    }
```

45.把数组排成最小的数

- 如何把数组合理地排序
- `Collections.sort(strs,(o1, o2)->(o1 + o2).compareTo(o2 + o1 ));`

```
/**
     执行耗时:7 ms,击败了54.20% 的Java用户
     内存消耗:38.1 MB,击败了55.27% 的Java用户
     */
    public String minNumber(int[] nums) {
        String[] strs = new String[nums.length];
        for(int i = 0 ; i < nums.length ; i++){
            strs[i] = String.valueOf(nums[i]);
        }
        //Collections.sort(strs,(o1, o2)->(o1 + o2).compareTo(o2 + o1 ));
        //  比如 o1 = 3 , o2 = 30
        //  330 > 303 所以 3 排在 30 后面
        quickSort(strs,0,strs.length-1);
        StringBuilder sb = new StringBuilder();
        for(String s : strs){
            sb.append(s);
        }
        return sb.toString();
    }


    void quickSort(String[] nums,int left,int right){
        if(left >= right){
```

```
            return;
        }
        int i = left,j = right;
        String temp = nums[left];
        while(i < j){
            while((nums[j] + nums[left]).compareTo(nums[left] + nums[j]) >=0
&& i < j)
                j--;
            while((nums[i] + nums[left]).compareTo(nums[left] + nums[i]) <=0
&& i < j)
                i++;
            String s = nums[i];
            nums[i] = nums[j];
            nums[j] = s;
        }
        nums[left] = nums[i];
        nums[i] = temp;
        quickSort(nums,left,i-1);
        quickSort(nums,i+1,right);
    }
```

46.把数字翻译成字符串

- 类似回溯？效果并不理想

```
/**
     *  回溯法 并不理想
     执行耗时:1 ms,击败了13.72% 的Java用户
     内存消耗:35.4 MB,击败了28.99% 的Java用户
     */
    public int translateNum(int num) {
        count = 0;
        dfs(String.valueOf(num),0);
        return count;
    }

    private void dfs(String num,int begin){
        if(begin == num.length()){
            count++;
            return;
        }
        int x = num.charAt(begin) - '0';
        if (x >= 0 && x <= 25) {
            if (begin + 1 <= num.length()) {
                dfs(num, begin + 1);
            }

      if (begin + 2 <= num.length()) {
            String s = num.substring(begin, begin + 2);
            int y = Integer.valueOf(s);
            if (y >= 10 && y <= 25) {
              dfs(num, begin + 2);
            }
```

```
        }
      }
    }
```

- 动态规划

```
/**
     * dp
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:35 MB,击败了89.86% 的Java用户
     */
    public int translateNum(int num) {
        String s = String.valueOf(num);
        int[] dp = new int[s.length() + 1];
        dp[0] = 1;
        dp[1] = 1;
        for(int i = 2 ; i <= s.length() ; i++){
            char c = s.charAt(i - 2);
            //String temp = s.substring(i - 2, i );
            //if(temp.compareTo("10") >= 0 && temp.compareTo("25") <= 0){
            if(c == '1' || (c == '2' && s.charAt(i - 1) <= '5')){
                dp[i] = dp[i - 2] + dp[i - 1] ;
            }
            else{
                dp[i] = dp[i - 1] ;
            }
        }
        return dp[s.length()];
    }
```

47.礼物的最大价值

- 动态规划
- 在外面增加一层0

```
    /**
     执行耗时:3 ms,击败了78.70% 的Java用户
     内存消耗:41.2 MB,击败了48.05% 的Java用户
     */
    public int maxValue(int[][] grid) {
        int row = grid.length;
        int col = grid[0].length;
        int[][] dp = new int[row + 1][col + 1];
        for(int i = 0 ; i <= row ; i++){
            for(int j = 0 ; j <= col ; j++){
                if(i == 0 || j == 0){
                    dp[i][j] = 0;
                }
                else {
                    dp[i][j] = grid[i - 1][j - 1] + Math.max(dp[i - 1]
[j],dp[i][j - 1]);
```

```
                }
            }
        }
        return dp[row][col];
    }
```

- 原地dp

```
/**
     执行耗时:2 ms,击败了97.87% 的Java用户
     内存消耗:41.1 MB,击败了71.91% 的Java用户
     */
    public int maxValue(int[][] grid) {
        int row = grid.length;
        int col = grid[0].length;
        for(int i = 1 ; i < col ; i++){
            grid[0][i] += grid[0][i - 1];
        }
        for(int i = 1 ; i < row ; i++){
            grid[i][0] += grid[i - 1][0];
        }
        for(int i = 1 ; i < row ; i++){
            for(int j = 1 ; j < col ; j++){
                grid[i][j] += Math.max(grid[i-1][j],grid[i][j-1]);
            }
        }
        return grid[row - 1][col - 1];

    }
```

48.最长不含重复字符的子字符串

- 滑动窗口
- 注意：调整左边边界的时候要一直往右找到第一个重复的字符

```
/**
     执行耗时:9 ms,击败了33.93% 的Java用户
     内存消耗:38.5 MB,击败了60.81% 的Java用户
     */
    public int lengthOfLongestSubstring(String s) {
        int left = 0;
        int res = 0;
        Set<Character> set = new HashSet<>();
        for(int i = 0 ; i < s.length(); i++){
            char c = s.charAt(i);
            // 这个位置要while不能if
            while(set.contains(c)){
                set.remove(s.charAt(left++));
            }
            set.add(c);
            res = Math.max(res,i - left + 1);
```

```
        }
        return res;
    }
```

- 类似动态规划的想法

```
/**
     执行耗时:9 ms,击败了33.93% 的Java用户
     内存消耗:38.5 MB,击败了63.18% 的Java用户
     */
    public int lengthOfLongestSubstring(String s) {
        Map<Character,Integer> map = new HashMap<>();
        int left = -1;
        int res =  0;
        for(int i = 0 ; i < s.length() ; i++){
            char c = s.charAt(i);
            if(map.containsKey(c)){
                // 这个地方与上面的while是一个意思
                left = Math.max(left,map.get(c));
            }
            map.put(c,i);
            res = Math.max(res,i - left);
        }
        return res;
    }
```

49.丑数

- 三指针
- 注意：存在 3 * 2 = 2 * 3 这种情况 所以指针往前推进的时候是三个`if`判断，而不是`if - else if`

```
/**
     执行耗时:2 ms,击败了99.14% 的Java用户
     内存消耗:37.6 MB,击败了40.91% 的Java用户
     */
    public int nthUglyNumber(int n) {
        int ptr1 = 0;
        int ptr2 = 0;
        int ptr3 = 0;
        int[] dp = new int[n];
        dp[0] = 1;
        for(int i = 1 ; i < n ; i++){
            int x1 = dp[ptr1] * 2;
            int x2 = dp[ptr2] * 3;
            int x3 = dp[ptr3] * 5;
            dp[i] = Math.min(x1,Math.min(x2,x3));
            // 这里不使用if - else if
            if(dp[i] == x1)
                ptr1++;
            if(dp[i] == x2)
                ptr2++;
```

```
            if(dp[i] == x3)
                ptr3++;
        }
        return dp[n - 1];
    }
```

50.第一个只出现一次的字符

- 两次遍历，用数组实现类似map的效果

```
/**
     执行耗时:8 ms,击败了77.05% 的Java用户
     内存消耗:38.8 MB,击败了62.04% 的Java用户
     */
    public char firstUniqChar(String s) {
        int[] res = new int[26];
        for(int i = 0 ; i < s.length() ; i++){
            res[s.charAt(i) - 'a']++;
        }

        for(int i = 0 ; i < s.length() ; i++){
            if(res[s.charAt(i) - 'a'] == 1)
                return s.charAt(i);
        }
        return ' ';
    }
```

52.两个链表的第一个公共节点

- 分别求出长度，长的链表先往前走差值

```
/**
     * 执行用时：1 ms, 在所有 Java 提交中击败了100.00%的用户
     内存消耗：41.1 MB, 在所有 Java 提交中击败了82.47%的用户
     */
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
          int lengthA = getLength(headA);
          int lengthB = getLength(headB);
          if(lengthA >lengthB){
              int step = lengthA -lengthB;
              while(step-- > 0){
                  headA = headA.next;
              }
          }
          else {
              int step = lengthB -lengthA;
              while(step-- > 0){
                  headB = headB.next;
              }
          }

            while(headA != null && headB != null && headA != headB){
```

```
                headA = headA.next;
                headB = headB.next;
            }
            return headA;
    }

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

- 浪漫相遇思路

```
/**
     执行用时：1 ms, 在所有 Java 提交中击败了100.00%的用户
     内存消耗：41.2 MB, 在所有 Java 提交中击败了67.13%的用户
     */
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
          ListNode h1 = headA;
          ListNode h2 = headB;
          while(h1 != h2){
              h1 = h1 == null ? headB : h1.next;
              h2 = h2 == null ? headA : h2.next;
          }
          return h1;
    }
```

53-I. 在排序数组中查找数字个数

- 二分，注意：在第一次二分完后如果mid指向的数不是target直接返回

```
/**
     执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
     内存消耗：41.4 MB, 在所有 Java 提交中击败了43.46%的用户
     */
    public int search(int[] nums, int target) {
        if(nums == null || nums.length == 0)
            return 0;
        int low = 0;
        int high = nums.length - 1;
        while(low < high){
            int mid = low + (high - low) / 2;
            if(nums[mid] >= target){
                high = mid;
            }
            else {
                low = mid + 1;
            }
```

```
        }

        // 精髓所在
        if(nums[low] != target)
            return 0;
        int l = low;

        high = nums.length;
        while(low < high){
            int mid = low + (high - low) / 2;
            if(nums[mid] <= target){
                low = mid + 1;
            }
            else{
                high = mid;
            }
        }
        int r = high;
        return r - l ;
    }
```

53-II 0~n-1中缺失的数字

- 二分法，找第一个索引与值不相等的位置
- 若`nums[m] = m`,则右子数组的首位元素一定在`[m + 1, j]`中,所以`i = m + 1`
- 若`nums[m] != m`,则左子数组的末位元素一定在`[i, m - 1]`中,所以`j = m - 1`

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:38.8 MB,击败了81.79% 的Java用户
     */
    public int missingNumber(int[] nums) {
        int i = 0, j = nums.length - 1;
        // 最终结果i、j分别指向右子数组首位与左子数组末位，所以结束时i>j
        while(i <= j) {
            int m = (i + j) / 2;
            if(nums[m] == m) i = m + 1;
            else j = m - 1;
        }
        return i;
    }
```

54.二叉搜索树的第K大节点

- 中根遍历递归

```
/**
     执行耗时:1 ms,击败了42.72% 的Java用户
```

```
     内存消耗:38.9 MB,击败了18.53% 的Java用户
     */
    List<Integer> res = new ArrayList<>();
    public int kthLargest(TreeNode root, int k) {
        midOrder(root);
        return res.get(res.size() - k);
    }

    private void midOrder(TreeNode treeNode){
        if(treeNode != null){
            midOrder(treeNode.left);
            res.add(treeNode.val);
            midOrder(treeNode.right);
        }
    }
```

- 中根遍历非递归
- 注意：`while`判断条件是`root != null || !stack.isEmpty()`之间是或

```
public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while(root != null || !stack.isEmpty()){
            while(root != null){
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
        return res;
    }
```

- 递归提前结束

```
    /**
    执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
    内存消耗：37.9 MB, 在所有 Java 提交中击败了96.85%的用户
    */
    int n ;
    int res;
    public int kthLargest(TreeNode root, int k) {
        n = k;
        midOrder(root);
        return res;
    }
    private void midOrder(TreeNode treeNode){
        if(treeNode != null){
            midOrder(treeNode.right);
            n--;
            if(n == 0){
                res = treeNode.val;
                return;
            }
            midOrder(treeNode.left);
```

```
        }
    }
```

- 从右往左遍历，非递归

```
/**
     执行耗时:1 ms,击败了42.72% 的Java用户
     内存消耗:38.2 MB,击败了71.38% 的Java用户
     */
    public int kthLargest(TreeNode root, int k) {
        Deque<TreeNode> stack = new LinkedList<>();
        stack.push(root);
        int count = 0;
        TreeNode temp = root;
        while(!stack.isEmpty()){
            while(temp!= null && temp.right != null){
                stack.push(temp.right);
                temp = temp.right;
            }
            temp = stack.poll();
            count++;
            if(count == k)
                return temp.val;
            temp = temp.left;
            if(temp != null){
                stack.push(temp);
            }
        }
        return -1;
    }
```

55.二叉树的深度

- 递归

```
  /**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:38.1 MB,击败了92.83% 的Java用户
     */
    public int maxDepth(TreeNode root) {
        if(root != null)
            return 1 + Math.max(maxDepth(root.left),maxDepth(root.right));
        else
            return 0;
    }
```

- 层序遍历

```
/**
     执行耗时:1 ms,击败了20.90% 的Java用户
     内存消耗:38.2 MB,击败了88.34% 的Java用户
     */
    public int maxDepth(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
```

```
        if(root == null)
            return 0;
        queue.offer(root);
        int height = 0;
        while(!queue.isEmpty()){
            height++;
            int size = queue.size();
            while(size-- > 0){
                TreeNode treeNode = queue.poll();
                if(treeNode.left != null)
                    queue.add(treeNode.left);
                if(treeNode.right != null)
                    queue.add(treeNode.right);
            }
        }
        return height;
    }
```

- dfs

```
/**
     执行耗时:0 ms,击败了100.00% 的Java用户
     内存消耗:38.3 MB,击败了77.23% 的Java用户
     */
    int res = 0;
    public int maxDepth(TreeNode root) {
        dfs(root,0);
        return res;
    }

    private void dfs(TreeNode root,int depth){
        if(root == null){
            res = Math.max(res,depth);
            return;
        }
        dfs(root.left,depth + 1);
        dfs(root.right,depth + 1);
    }
```

55. 判断是不是二叉平衡树

- 层序 + 求树高，效率是真的拉胯，我太笨了

```
/**
     执行耗时:2 ms,击败了7.21% 的Java用户
     内存消耗:38.1 MB,击败了95.90% 的Java用户
     */
    public boolean isBalanced(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        if(root == null)
            return true;
        queue.offer(root);
        TreeNode temp;
        while (!queue.isEmpty()){
```

```
            int size = queue.size();
            while (size-- > 0){
                temp = queue.poll();
                if(Math.abs(getHeight(temp.right) - getHeight(temp.left)) >
1)
                    return false;
                if(temp.left != null)
                    queue.offer(temp.left);
                if(temp.right != null)
                    queue.offer(temp.right);
            }
        }
        return true;
    }

    private int getHeight(TreeNode root){
        if(root != null)
            return 1 + Math.max(getHeight(root.left),getHeight(root.right));
        else
            return 0;
    }
```

- 左右条件判断 + 求树高
- 这边的思路与第26类似

```
/**
     执行耗时:1 ms,击败了100.00% 的Java用户
     内存消耗:38.9 MB,击败了10.57% 的Java用户
     */
    public boolean isBalanced(TreeNode root) {
        if(root == null)
            return true;
        if(!isBalanced(root.left) ||!isBalanced(root.right))
            return false;
        return Math.abs(getHeight(root.right) - getHeight(root.left)) <= 1;
    }

    private int getHeight(TreeNode root){
        if(root != null)
            return 1 + Math.max(getHeight(root.left),getHeight(root.right));
        else
            return 0;
    }
```

- 可以借鉴的方法

```
/**
     执行耗时:1 ms,击败了100.00% 的Java用户
     内存消耗:38 MB,击败了98.71% 的Java用户
     */
    public boolean isBalanced(TreeNode root) {
        return getHeight(root) >= 0;
    }

    private int getHeight(TreeNode root){
        if(root == null)
```

```
            return 0;
        int left = getHeight(root.left);
        int right = getHeight(root.right);
        if(left >=0 && right >= 0 && Math.abs(right - left) <= 1)
            return Math.max(right , left) + 1;
        // -1 表示不平衡
        return -1;
    }
```