LC-206 反转链表
LC-146 LRU
LC-215 数组中第K个最大元素
LC-25 K个一组的反转链表
LC-3 无重复字符的最长子串
LC-121 买卖股票的最佳时机
LC-15 三数之和
LC-103 二叉树的锯齿形层序遍历
LC-160 相交链表
LC-21 合并两个有序链表
LC-199 二叉树的右视图
LC-102 二叉树的层序遍历
LC-141 环形链表
LC-142 环形链表2
LC-92 反转特定位置链表2
LC-33 搜索旋转排序数组
LC-42 接雨水
LC-1 两数之和
LC-2 两数相加
LC-415 字符串相加
LC-54 螺旋矩阵
LC-23 合并K个升序链表
LC-236 二叉树的最近公共祖先
LC-155 最小栈
LC-69 x的平方根
LC-20 有效的括号
LC-234 回文链表
LC-94 二叉树中序遍历
LC-5 最长回文子串
LC-31 下一个排列
LC-72 编辑距离
LC-165 比较版本号
LC-912 排序数组
LC-88 合并两个有序数组
LC-704 二分查找
LC-232 用栈实现队列
LC-46 全排列
LC-53 最大子序和
LC-8 字符串转换整数
LC-124 二叉树中的最大路径和
LC-143 重排链表
LC-300 最长递增子序列
LC-470 用Rand7()实现Rand10()
LC-200 岛屿数量
LC-82 删除排序链表中的重复元素2
LC-221 最大正方形
LC-227 基本计算器2
LC-48 旋转图像
LC-113 路经总和2
LC-1353 最多可以参加的会议数目
LC-138 复制带随机指针的链表
LC-974 和可被K整除的子数组
LC-662 二叉树的最大宽度
LC-110 平衡二叉树
LC-198 打家劫舍

LC-213 打家劫舍2
LC-287 寻找重复数
面试题 02.04. 分割链表
课程表

## LC-206 反转链表

```
public ListNode reverseList(ListNode head) {
    ListNode pre = null;
    ListNode cur = head;
    while(cur != null){
        ListNode temp = cur.next;
        cur.next = pre;
        pre = cur;
        cur = temp;
    }
    return pre;
}
```

## LC-146 LRU

相关分析

- O(1)的插入与获取

  链表(插入)+哈希(获取)

- 最近最少使用

  双向链表，头部为最近使用，尾部为最近最少使用

- 虚拟头尾指针

  添加/删除节点不需考虑相邻节点

相关流程

- get()

  从哈希获取key对应的node

  - node不存在，返回-1
  - node存在，将node移到头部，返回value

- put()

  从哈希获取key对应的node

  - node不存在，新建节点，插入头部，添加哈希
    - 节点过多，删除尾节点，删除哈希
  - node存在，将node移到头部，更新value

```
class LRUCache {

    class Node {  // 双向链表节点
        int key;
        int value;
        Node pre;
        Node next;
```

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

    private Map<Integer, Node> cache = new HashMap<Integer, Node>();
    private int size;
    private int capacity;
    private Node head, tail;  // 虚拟头尾指针

    public LRUCache(int capacity) {
        this.size = 0;
        this.capacity = capacity;
        head = new Node();
        tail = new Node();
        head.next = tail;
        tail.pre = head;
    }

    public int get(int key) {
        // 判断key
        Node node = cache.get(key);

        if (node == null) return -1;  // key不存在 返回-1
        else{
            // key存在 将对应节点移到头部 返回value
            node.pre.next = node.next;
            node.next.pre = node.pre;

            node.pre = head;
            node.next = head.next;
            head.next.pre = node;
            head.next = node;

            return node.value;
        }
    }

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

        if(node == null){
            // key不存在
            // 创建新节点 在头部添加节点 在哈希表添加节点
            // 判断节点数是否超出容量 超出则删除尾部节点和哈希表对应项

            Node newNode = new Node(key, value);

            newNode.pre = head;
            newNode.next = head.next;
            head.next.pre = newNode;
            head.next = newNode;

            cache.put(key, newNode);

            size++;
            if(size > capacity){
                Node deleteNode = tail.pre;
```

```
                deleteNode.pre.next = deleteNode.next;
                deleteNode.next.pre = deleteNode.pre;

                cache.remove(deleteNode.key);

                size--;
            }
        }else{
            // key存在 将对应节点移到头部 更新value值
            node.pre.next = node.next;
            node.next.pre = node.pre;

            node.pre = head;
            node.next = head.next;
            head.next.pre = node;
            head.next = node;

            node.value = value;
        }
    }
}
```

## LC-215 数组中第K个最大元素

基于快速排序分区思想的快速搜索

```
public int findKthLargest(int[] nums, int k) {
    int index = nums.length - k;  // 第K个最大元素在数组的索引值
    return quickSearch(nums,0,nums.length-1,index);
}

private int quickSearch(int[] nums, int left, int right, int k){
    int partition = partition(nums,left,right);

    if(partition == k) return nums[partition];
    if(partition < k) return quickSearch(nums,partition+1,right,k);
    return quickSearch(nums,left,partition-1,k);
}

private int partition(int[] nums, int left, int right){
    int base = nums[left];
    int i = left;
    int j = right;

    while(true){
        while(nums[i] <= base && i < right) i++;
        while(nums[j] >= base && j > left) j--;
        if(i >= j) break;
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    nums[left] = nums[j];
    nums[j] = base;
```

```
    return j;
}
```

## LC-25 K个一组的反转链表

增加虚拟头节点，记录翻转后的头节点和化归第一个翻转链表

```
public ListNode reverseKGroup(ListNode head, int k) {
    ListNode vhead = new ListNode();  // 虚拟头节点
    vhead.next = head;

    ListNode left = vhead;
    ListNode right = vhead;

    // left -> (i)(j)ListNode(i) -> (j)right
    while(true){
        // move right
        for(int i = 0; i < k; i++){
            right = right.next;
            if(right == null) return vhead.next;
        }
        right = right.next;

        ListNode i = left.next;
        ListNode j = i.next;

        while(j != right){
            ListNode temp = j.next;
            j.next = i;
            i = j;
            j = temp;
        }

        // 翻转后 头节点-i 尾节点-left.next
        left.next.next = right;  // 处理尾节点
        right = left.next;  // 记录尾节点 同时也是下次翻转的起点
        left.next = i;  // 处理头节点
        left = right;  // 初始化下次翻转
    }
}
```

## LC-3 无重复字符的最长子串

滑动窗口

```
public int lengthOfLongestSubstring(String s) {
    int left = 0;
    int right = 0;
    int res = 0;

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

    while(right < s.length()){
        Character newChar = s.charAt(right);
        right++;
        m.put(newChar, m.getOrDefault(newChar, 0) + 1);
```

```
        while(m.get(newChar) > 1){
            Character oldChar = s.charAt(left);
            left++;
            m.put(oldChar, m.get(oldChar) - 1);
        }

        res = Math.max(res, right - left);
    }

    return res;
}
```

### LC-121 买卖股票的最佳时机

动态规划，发现当前天的最大利润只跟前一天的最大利润和当前天卖出的利润有关，状态压缩至两个变量

```
public int maxProfit(int[] prices) {
    int maxProfit = 0;
    int minPrices = prices[0];

    for(int price : prices){
        maxProfit = Math.max(maxProfit, price - minPrices);
        minPrices = Math.min(minPrices, price);
    }

    return maxProfit;
}
```

### LC-15 三数之和

不使用Set去重的情况下，需要跳过重复的元素，提升时间复杂度

```
public List<List<Integer>> threeSum(int[] nums) {

    if(nums.length < 3) return new ArrayList();

    // 排序 + 双指针逼近 + Set去重
    Arrays.sort(nums);
    Set<List<Integer>> res = new HashSet();

    for(int i = 0; i < nums.length - 2; i++){
        if(nums[i] > 0) break;
        // if (i > 0 && nums[i] == nums[i - 1]) continue; // 去重

        // 双指针逼近
        int left = i + 1;
        int right = nums.length - 1;

        while(left < right){
            if(nums[left] + nums[right] + nums[i] == 0){
                res.add(new
ArrayList(Arrays.asList(nums[i],nums[left],nums[right])));
                left++;
                right--;
```

```
                // while (left < right && nums[left] == nums[left - 1]) left++;
// 去重
                // while (left < right && nums[right] == nums[right + 1]) right--; // 去重
            }
            else if (nums[left] + nums[right] + nums[i] > 0) right--;
            else left++;
        }
    }

    return new ArrayList<List<Integer>>(res);
}
```

## LC-103 二叉树的锯齿形层序遍历

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    if(root == null) return new ArrayList();

    Queue<TreeNode> q = new LinkedList<>();
    List<List<Integer>> res = new ArrayList();

    q.offer(root);
    int depth = 1;

    while(q.size() > 0){
        int size = q.size();
        List<Integer> curRes = new ArrayList();

        for(int i = 0; i < size; i++){
            TreeNode curNode = q.poll();
            if(curNode.left != null) q.offer(curNode.left);
            if(curNode.right != null) q.offer(curNode.right);

            if((depth & 1) == 1) curRes.add(curNode.val);  // 奇数层 正向
            else curRes.add(0, curNode.val);  // 偶数层 反向
        }

        res.add(curRes);
        depth++;
    }

    return res;
}
```

## LC-160 相交链表

两轮遍历

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    ListNode a = headA;
    ListNode b = headB;

    while(a != b){
        if(a != null) a = a.next;
        else a = headB;
        if(b != null) b = b.next;
```

```
            else b = headA;
        }

        // 停止条件 a==b
        // 1. ab等长 则一轮结束 要么相交 要么a==b==null
        // 2. ab不等长 则两轮结束 要么相交 要么a==b==null
        return a;
    }
```

## LC-21 合并两个有序链表

虚拟头节点

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode vHead = new ListNode();  // 虚拟头节点
    ListNode cur = vHead;

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

    if(l1 != null) cur.next = l1;
    if(l2 != null) cur.next = l2;

    return vHead.next;
}
```

## LC-199 二叉树的右视图

DFS不行，[1,2,3,4] -> [1,3,4] 会漏掉4

BFS，Deque当作队列(addLast/pollFirst)

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> res = new ArrayList();

    if(root == null) return res;

    // bfs 队列-addLast/pollFirst 每层最后一个记录下
    Deque<TreeNode> deque = new LinkedList();

    deque.addLast(root);

    while(deque.size() > 0){
        int size = deque.size();
        for(int i = 0; i < size; i++){
            TreeNode tn = deque.pollFirst();
            if(i == size - 1) res.add(tn.val);
            if(tn.left != null) deque.addLast(tn.left);
```

```
            if(tn.right != null) deque.addLast(tn.right);
        }
    }

    return res;
}
```

### LC-102 二叉树的层序遍历

BFS，Deque当作队列(addLast/pollFirst)

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList();
    if(root == null) return res;

    Deque<TreeNode> deque = new LinkedList();
    deque.addLast(root);

    while(deque.size() > 0){
        int size = deque.size();
        List<Integer> curRes = new ArrayList();
        for(int i = 0; i < size; i++){
            TreeNode tn = deque.pollFirst();
            curRes.add(tn.val);
            if(tn.left != null) deque.addLast(tn.left);
            if(tn.right != null) deque.addLast(tn.right);
        }
        res.add(curRes);
    }

    return res;
}
```

### LC-141 环形链表

- 哈希表(HashSet)
    - 每次将节点放入Set中，若失败则表明访问了相同节点
- 快慢指针
    - 有环，快指针会追上慢指针
    - 无环，快指针会到达终点(null)

```
// 哈希表
public boolean hasCycle(ListNode head) {
    Set<ListNode> set = new HashSet();
    while(head != null){
        if(set.add(head)) head = head.next;  // add()返回false表示Set有相同元素
        else return true;
    }
    return false;
}
// 快慢指针
public boolean hasCycle(ListNode head) {
    if(head == null) return false;

    // 初始时快指针走两步 即到达head.next 慢指针走一步 即到达head
```

```
    ListNode fast = head.next;
    ListNode slow = head;

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

## LC-142 环形链表2

- 快慢指针

  设链表有环，环外节点a个节点，坏内节点b个节点

  第一轮相遇时，设快指针走了f个节点，慢指针走了s个节点

  - f = 2s
  - f-s = nb(n为正整数)

  可得s为nb，此时若慢指针再走a个节点，会正好停在环内节点起点

  第二轮慢指针初始化到头节点，快慢指针同时走，相遇时即快指针走了a，慢指针走了a+nb

```
public ListNode detectCycle(ListNode head) {
    if(head == null || head.next == null) return null;

    // 初始时快指针走两步 即到达head.next 慢指针走一步 即到达head
    ListNode fast = head.next;
    ListNode slow = head;

    // 第一轮相遇时 slow走了nb
    while(true){
        if(fast == null || fast.next == null || fast.next.next == null ) return
null;
        if(fast == slow) break;
        fast = fast.next.next;
        slow = slow.next;
    }

    // 第二轮相遇时 slow走了a+nb 停在环内节点起点
    fast = head;
    slow = slow.next;
    while(fast != slow){
        fast = fast.next;
        slow = slow.next;
    }

    return slow;
}
```

## LC-92 反转特定位置链表2

```
public ListNode reverseBetween(ListNode head, int left, int right) {
    // left => (i)(j)反转子链表(i) => (j)right
    ListNode vHead = new ListNode();
    vHead.next = head;

    ListNode leftLN = vHead;
    ListNode rightLN = vHead;

    // left 走 left-1 次
    for(int i = 0; i < left - 1; i++){
        leftLN = leftLN.next;
    }

    // right 走 right+1 次
    for(int i = 0; i < right + 1; i++){
        rightLN = rightLN.next;
    }

    // 子链表反转
    ListNode i = leftLN.next;
    ListNode j = i.next;
    while(j != rightLN){
        ListNode temp = j.next;
        j.next = i;
        i = j;
        j = temp;
    }

    // 处理头尾
    leftLN.next.next = rightLN;
    // right = left.next;  // k个反转下次反转起点
    leftLN.next = i;
    // left = right;  // k个反转下次反转起点

    return vHead.next;
}
```

## LC-33 搜索旋转排序数组

- 找到旋转位置后转换成二分查找，效率较低
- 直接二分查找，分段讨论
    - 判断left与mid是否在同一段，可推理[left, mid]单调递增还是[mid, right]单调递增
    - 判断target是否处于递增区间内

```
// 找到旋转位置后转换成二分查找
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;

    // 找到旋转位置
    int mid = 0;
    while(mid < right && nums[mid] < nums[mid + 1]) mid++;
```

```
    // 找到二分查找范围
    if(nums[left] > target) left = mid + 1;
    else right = mid;

    // 二分查找
    while(left <= right){
        mid = (left + right) / 2;
        if(nums[mid] > target) right = mid - 1;
        else if (nums[mid] < target) left = mid + 1;
        else return mid;
    }

    return -1;
}

// 直接二分
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;

    while(left <= right){
        int mid = left + (right - left) / 2;

        if(target == nums[mid]) return mid;

        if(nums[left] <= nums[mid]){  // [left, mid]区间递增
            if(target >= nums[left] && target < nums[mid]) right = mid - 1;  //
[left, target, mid)
            else left = mid + 1; // (mid, target right]
        }else{  // [mid, right]区间递增
            if(target > nums[mid] && target <= nums[right]) left = mid + 1;  //
(mid, target, right]
            else right = mid - 1;  // [left, target, mid)
        }
    }

    return - 1;
}
```

## LC-42 接雨水

- 暴力解法

  求每一列的雨水，等于每列左边和右边最高的墙的较矮的那个墙与自身的差值

- 优化。。。

```
public int trap(int[] height) {
    int res = 0;

    // 最两端的列不考虑 不存水
    for(int i = 1; i < height.length - 1; i++){
        // 左边最高的墙
        int maxLeft = 0;
        for(int j = i - 1; j >= 0; j--) maxLeft = Math.max(maxLeft,height[j]);

        // 右边最高的墙
        int maxRight = 0;
```

```
        for(int j = i + 1; j < height.length; j++) maxRight =
Math.max(maxRight,height[j]);

        // 左右最高墙的较矮的墙
        int minHeight = Math.min(maxLeft,maxRight);

        // 比自身高 则可存水
        if(minHeight > height[i]) res = res + minHeight - height[i];
    }
    return res;
}
```

## LC-1 两数之和

哈希表存储值和索引

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer,Integer> map = new HashMap<>();
    for(int i = 0 ; i < nums.length; i++){
        Integer index = map.get(target - nums[i]);
        if(index != null) return new int[]{i,index};
        else map.put(nums[i],i);
    }
    return new int[0];
}
```

## LC-2 两数相加

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode vHead = new ListNode();
    ListNode cur = vHead;

    int carry = 0;

    while(l1 != null && l2 != null){
        int sum = l1.val + l2.val + carry;
        carry = sum / 10;

        cur.next = new ListNode(sum % 10);

        cur = cur.next;
        l1 = l1.next;
        l2 = l2.next;
    }

    while(l1 != null){
        int sum = l1.val + carry;
        carry = sum / 10;

        cur.next = new ListNode(sum % 10);

        cur = cur.next;
        l1 = l1.next;
    }

    while(l2 != null){
```

```
            int sum = l2.val + carry;
            carry = sum / 10;

            cur.next = new ListNode(sum % 10);

            cur = cur.next;
            l2 = l2.next;
        }

        if(carry > 0) cur.next = new ListNode(carry);  // 最后还有进位要处理

        return vHead.next;
    }
```

## LC-415 字符串相加

类似上题的思路

- StringBuilder从头插入，insert(0,char)
- char转int，(int)char - (int)'0'
- int转char，(char)(int+'0')

```
public String addStrings(String num1, String num2) {

    int i = num1.length() - 1;
    int j = num2.length() - 1;

    StringBuilder res = new StringBuilder();

    int carry = 0;
    while(i >= 0 || j >= 0){
        int sum = carry;
        if(i >= 0){
            sum = sum + (int) num1.charAt(i) - (int) '0';
            i--;
        }
        if(j >= 0){
            sum = sum + (int) num2.charAt(j) - (int) '0';
            j--;
        }
        carry = sum / 10;
        res.insert(0,sum % 10);
    }

    if(carry > 0) res.insert(0,carry);

    return res.toString();
}
```

## LC-54 螺旋矩阵

收缩边界法

```
public List<Integer> spiralOrder(int[][] matrix) {
    int l = 0;
    int r = matrix[0].length - 1;
```

```
    int t = 0;
    int b = matrix.length - 1;

    List<Integer> res = new ArrayList();

    while(true){
        for(int i = l; i <= r; i++) res.add(matrix[t][i]);
        t++;
        if(t > b) break;

        for(int i = t; i <= b; i++) res.add(matrix[i][r]);
        r--;
        if(r < l) break;

        for(int i = r; i >= l; i--) res.add(matrix[b][i]);
        b--;
        if(b < t) break;

        for(int i = b; i >= t; i--) res.add(matrix[i][l]);
        l++;
        if(l > r) break;
    }

    return res;
}
```

## LC-23 合并K个升序链表

- 列表排序

  遍历所有节点放入队列，排序队列，构建有序链表

- 分治法

  类似归并排序，先拆分成最小单位两个链表，最后再不断合并

- 优先队列/小顶堆

  类似BFS，将所有节点放入优先队列中，每次获取其中最小值的节点，构建有序链表

```
// 列表排序
public ListNode mergeKLists(ListNode[] lists) {
    // 遍历所有节点放入队列
    List<ListNode> l = new ArrayList();
    for(ListNode ln : lists){
        while(ln != null){
            l.add(ln);
            ln = ln.next;
        }
    }
    // 排序队列
    Collections.sort(l, (x,y)->x.val-y.val);
    // 构建有序链表
    ListNode vHead = new ListNode();
    ListNode cur = vHead;
    for(ListNode ln : l){
        cur.next = ln;
        cur = cur.next;
    }
    return vHead.next;
```

```
}
// 优先队列/小顶堆
public ListNode mergeKLists(ListNode[] lists) {
    // (x,y)->x.val-y.val 递增队列
    PriorityQueue<ListNode> pq = new PriorityQueue<>((x,y)->x.val-y.val);

    // 将所有节点放入优先队列中
    for(ListNode ln : lists){
        if(ln != null) pq.add(ln);
    }

    ListNode vHead = new ListNode();
    ListNode cur = vHead;

    while(pq.size() > 0){
        ListNode ln = pq.poll();  // 获取最小值的节点
        cur.next = ln;
        cur = cur.next;
        if(ln.next != null) pq.add(ln.next);  // 放入下一个节点
    }

    return vHead.next;
}
```

## LC-236 二叉树的最近公共祖先

若root为p、q的最近公共祖先，则有两种情况

1. root为p/q，q/p在其子树上
2. root不为p/q，p/q分别在左右子树上

dfs后续遍历，对于节点root，检测左右子树是否存在p、q节点

- 左右子树均含p、q节点，则满足情况2，root即为最近公共祖先
- 左右子树均不含p、q节点，则root不为所求，返回null
- 左/右子树其中一个子树含p、q节点，则可能满足情况1，向上回溯是否满足情况2的情况
    - 若向上回溯一直未满足情况2，则满足情况1，先遍历到的p、q节点即为所求

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    return dfs(root, p, q);
}

private TreeNode dfs(TreeNode root, TreeNode p, TreeNode q){
    if(root == null) return null;
    if(root == p || root == q) return root;

    // 检测左/右子树是否存在pq
    TreeNode left = dfs(root.left, p, q);
    TreeNode right = dfs(root.right, p, q);

    // 左右子树均含p、q节点
    if(left != null && right != null) return root;

    // 左右子树其中一个子树含p、q节点
    if(left != null && right == null) return left;
    if(left == null && right != null) return right;
```

```
    // 左右子树均不含p、q节点
    return null;  // if(left == null && right == null)
}
```

## LC-155 最小栈

- 辅助栈存储最小值
- 栈存储值与最小值的差值(节省空间)

```
// 辅助栈
Stack<Integer> data;
Stack<Integer> minData;

public MinStack() {
    data = new Stack();
    minData = new Stack();
}

public void push(int val) {
    data.push(val);
    if(minData.isEmpty()){
        minData.push(val);
    }else{
        minData.push(Math.min(val,minData.peek()));
    }
}

public void pop() {
    data.pop();
    minData.pop();
}

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

public int getMin() {
    return minData.peek();
}

// 栈存储值与最小值的差值
Integer min;
Stack<Long> data;

public MinStack() {
    data = new Stack<>();
}

// 栈值 =  x - 当前栈最小值
public void push(int x) {
    if(data.isEmpty()){
        // 第一个值
        min = x;
        data.push(0L);
    }else{
        // 非第一个值
```

```
        if(x < min){
            data.push(Long.valueOf(x) - min);
            min = x;
        }else{
            data.push(Long.valueOf(x) - min);
        }
    }
}

public void pop() {
    Long diff = data.pop();
    if(diff >= 0){
        // return (int)(diff + min);
    }else{
        int tempMin = min;
        min = (int)(min - diff);
        // return tempMin;
    }

}

public int top() {
    Long diff = data.peek();
    if(diff >= 0) return (int)(diff + min);
    else{
        return min;
    }
}
```

## LC-69 x的平方根

- 二分法
- 牛顿迭代法

  求x的平方根s，可通过定义初始值res=x，不断通过res=res+x/res逼近s

```
// 二分法
public int mySqrt(int x) {
    int l = 0;
    int r = x;
    int res = -1;
    while(l <= r){
        int mid = l + (r - l) / 2;
        if((long) mid * mid <= x){
            res = mid;
            l = mid + 1;
        }else r = mid - 1;
    }
    return res;
}
// 牛顿迭代法
public double sqrts(double x){
    double res = (x + s / x) / 2;
    if ((int)res == (int)x) return x;
    else return sqrts(res);
}
```

## LC-20 有效的括号

栈存储，保证括号位置正确

```
public boolean isValid(String s) {
    // 1. HashMap 存储对应关系
    Map<Character, Character> m = new HashMap();
    m.put('(', ')');
    m.put('{', '}');
    m.put('[', ']');

    // 2. 栈实现位置对应匹配
    Stack<Character> stack = new Stack<>();

    for(char c : s.toCharArray()){
        if(m.get(c) != null) stack.push(c);
        else if(stack.size() == 0 || m.get(stack.pop()) != c) return false;
    }

    return stack.size() == 0;
}
```

## LC-234 回文链表

- 数组模拟，空间复杂度较高
- 快慢指针，链表反转，空间复杂度高

```
// 数组模拟
public boolean isPalindrome(ListNode head) {
    List<ListNode> l = new ArrayList();

    while(head != null){
        l.add(head);
        head = head.next;
    }

    int left = 0;
    int right = l.size() - 1;
    while(left < right){
        if(l.get(left).val != l.get(right).val) return false;
        left++;
        right--;
    }

    return true;
}

// 快慢指针 反转链表
public boolean isPalindrome(ListNode head) {
    if(head == null || head.next == null) return true;

    ListNode slow = head;
    ListNode fast = head.next;
    ListNode slowNext = slow.next;

    while(fast != null && fast.next != null){
        ListNode temp = slow;
```

```
        slow = slowNext;
        slowNext = slowNext.next;
        fast = fast.next.next;
        slow.next = temp;
    }

    ListNode left;
    ListNode right;

    // 1 2 3 4 slow=2 fast=4 偶数
    // 1 2 3 4 5 slow=3 fast=6(null) 奇数
    if(fast != null){ // 偶数
        left = slow;
        right = slowNext;
    }else{  // 奇数
        left = slow.next;
        right = slowNext;
    }

    while(right != null){
        if(left.val != right.val) return false;
        left = left.next;
        right = right.next;
    }

    return true;
}
```

## LC-94 二叉树中序遍历

- 递归
- 迭代，利用栈的特性

```
// 迭代
public List<Integer> inorderTraversal(TreeNode root) {
    Stack<TreeNode> s = new Stack();
    List<Integer> res = new ArrayList();

    while(root != null || s.size() != 0){
        // 左
        while(root != null){
            s.push(root);
            root = root.left;
        }
        root = s.pop();
        // 根
        res.add(root.val);
        // 右
        root = root.right;
    }
    return res;
}
```

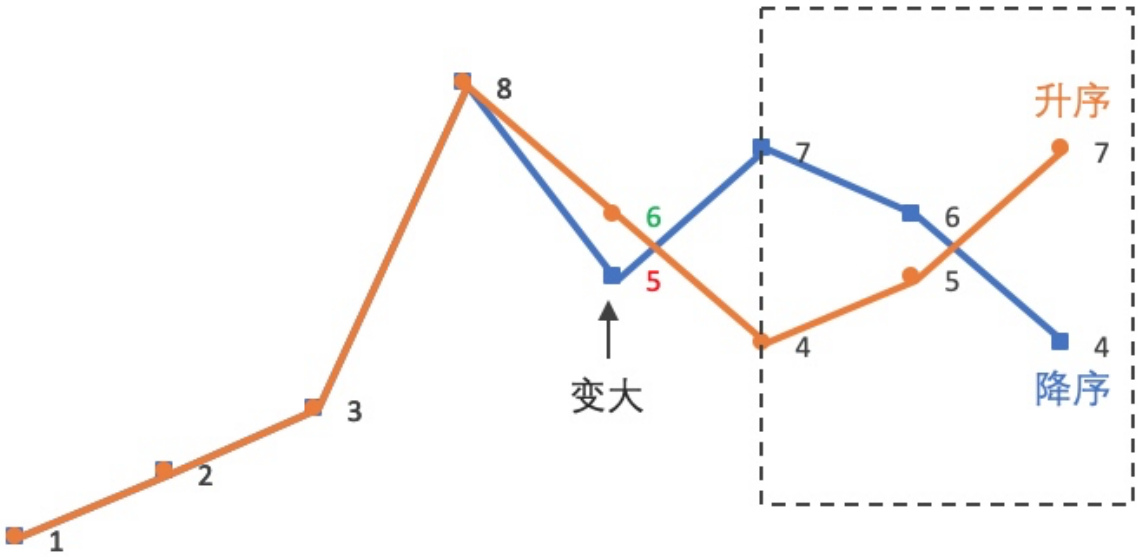
## LC-5 最长回文子串

遍历每个字符，双指针中心扩散，考虑中心是字符/间隙

```
public String longestPalindrome(String s) {
    if(s.length() <= 1) return s;

    String res = "";
    int left, right;
    String curRes;

    for(int i = 0; i < s.length(); i++){
        // 中心是字符
        left = i;
        right = i;
        while(left >= 0 && right <= s.length() - 1 && s.charAt(left) ==
s.charAt(right)){
            left--;
            right++;
        }
        curRes = s.substring(left + 1, right);
        if(curRes.length() > res.length()) res = curRes;

        // 中心是间隙
        left = i;
        right = i + 1;
        while(left >= 0 && right <= s.length() - 1 && s.charAt(left) ==
s.charAt(right)){
            left--;
            right++;
        }
        curRes = s.substring(left + 1, right);
        if(curRes.length() > res.length()) res = curRes;
    }

    return res;
}
```

## LC-31 下一个排列

- 从后往前，找到后排列的变小节点
- 变小节点与后排列的降序中的大于变小节点的最小值交换
- 后排列重新排为升序




```java
class Solution {
    public void nextPermutation(int[] nums) {
        if(nums.length <= 2){
            change(nums, 0, nums.length - 1);
            return;
        }

        // 从后向前 找到 nums[i] < nums[i+1]
        int i = nums.length - 2;
        while(i >= 0 && nums[i] >= nums[i + 1]) i--;

        // 若是最大序列 反转数组
        if(i == -1){
            Arrays.sort(nums);
            return;
        }

        // [i + 1, end]降序中刚好大于nums[i]的nums[index]
        int index = i + 1;
        while(index <= nums.length - 1 && nums[index] > nums[i]) index++;
        index--;

        // 交换
        change(nums, i, index);

        // 反转[i + 1, end]
        Arrays.sort(nums, i + 1, nums.length);
        return;
    }

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

## LC-72 编辑距离

**动态规划：**

`dp[i][j]` 代表 `word1` 到 `i` 位置转换成 `word2` 到 `j` 位置需要最少步数

所以，

当 `word1[i] == word2[j]`，`dp[i][j] = dp[i-1][j-1]`；

当 `word1[i] != word2[j]`，`dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1`

其中，`dp[i-1][j-1]` 表示替换操作，`dp[i-1][j]` 表示删除操作，`dp[i][j-1]` 表示插入操作。

注意，针对第一行，第一列要单独考虑，我们引入 `''` 下图所示：

| | '' | r | o | s |
|---|---|---|---|---|
| '' | 0 | 1 | 2 | 3 |
| h | 1 | 1 | 2 | 3 |
| o | 2 | 2 | 1 | 2 |
| r | 3 | 2 | 2 | 2 |
| s | 4 | 3 | 3 | 2 |
| e | 5 | 4 | 4 | 3 |

第一行，是 `word1` 为空变成 `word2` 最少步数，就是插入操作

第一列，是 `word2` 为空，需要的最少步数，就是删除操作

```
public int minDistance(String word1, String word2) {
    int n1 = word1.length();  // horse 5
    int n2 = word2.length();  // ros 3

    int[][] dp = new int[n1 + 1][n2 + 1];  // 6 * 4

    for(int i = 0; i < n1 + 1; i++) dp[i][0] = i;  // 第一列
    for(int i = 0; i < n2 + 1; i++) dp[0][i] = i;  // 第一行

    for(int i = 1; i < n1 + 1; i++){
        for(int j = 1; j < n2 + 1; 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-1][j], dp[i][j-1]), dp[i-1]
[j-1]) + 1;
        }
    }

    return dp[n1][n2];
}
```

## LC-165 比较版本号

双指针比较

```
public int compareVersion(String version1, String version2) {
    int size1 = version1.length();
    int size2 = version2.length();
    int v1 = 0;
    int v2 = 0;
```

```
    while(v1 < size1 || v2 < size2){
        // 计算每个版本号大小
        int sum1 = 0;
        while(v1 < size1 && version1.charAt(v1) != '.'){
            sum1 = sum1 * 10 + (int)version1.charAt(v1) - (int)'0';
            v1++;
        }

        int sum2 = 0;
        while(v2 < size2 && version2.charAt(v2) != '.'){
            sum2 = sum2 * 10 + (int)version2.charAt(v2) - (int)'0';
            v2++;
        }

        if(sum1 != sum2) return sum1 > sum2 ? 1 : -1;
        else {  // 相等跳过.继续比较
            v1++;
            v2++;
        }
    }
    return 0;
}
```

## LC-912 排序数组

- 快排
- 归并排序
    - 初始化临时数组
    - 划分数组
    - 合并数组

```
// 快排
public int[] sortArray(int[] nums) {
    quickSort(nums, 0, nums.length - 1);
    return nums;
}

private void quickSort(int[] nums, int l, int r){
    if(l >= r) return;
    int p = partition(nums, l, r);
    quickSort(nums, l, p - 1);
    quickSort(nums, p + 1, r);
}

private int partition(int[] nums, int l, int r){
    int base = nums[l];
    int i = l;
    int j = r;

    while(true){
        while(i < r && nums[i] <= base) i++;
        while(j > l && nums[j] >= base) j--;
        if(i >= j) break;
        int temp = nums[i];
        nums[i] = nums[j];
```

```
            nums[j] = temp;
        }

        // 由于nums[j]小于base 因此与base交换 放到数组前部
        nums[l] = nums[j];
        nums[j] = base;

        return j;
    }
    // 归并排序
    int[] temp;

    public int[] sortArray(int[] nums) {
        temp = new int[nums.length];  // 初始化临时数组
        mergeSort(nums, 0, nums.length - 1);
        return nums;
    }

    // 划分数组
    private void mergeSort(int[] nums, int l, int r){
        if(l >= r) return;

        int mid = (l + r) / 2;
        mergeSort(nums, l, mid);
        mergeSort(nums, mid + 1, r);

        merge(nums, l, mid, r);
    }

    // 合并数组
    private void merge(int[] nums, int l, int mid, int r){
        // 将两个有序数组复制至临时数组中
        for(int k = l; k <= r; k++){
            temp[k] = nums[k];
        }

        int i = l;
        int j = mid + 1;
        int index = l;

        // 将小的元素放回原数组
        while(i <= mid && j <= r){
            // 先判断左边的 保持稳定
            if(temp[i] <= temp[j]) nums[index++] = temp[i++];
            else nums[index++] = temp[j++];
        }

        // 将剩下的元素放回原数组
        if(i > mid){
            while(j <= r) nums[index++] = temp[j++];
        }
        if(j > r){
            while(i <= mid) nums[index++] = temp[i++];
        }
    }
```

## LC-88 合并两个有序数组

双指针从后往前合并

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
    int i = m - 1;
    int j = n -1;
    int index = m + n -1;

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

    while(i >= 0){
        nums1[index--] = nums1[i--];
    }

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

    return;
}
```

## LC-704 二分查找

```
public int search(int[] nums, int target) {
    int l = 0;
    int r = nums.length - 1;

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

    return -1;
}
```

## LC-232 用栈实现队列

```
Stack<Integer> s1;
Stack<Integer> s2;

public MyQueue() {
    s1 = new Stack();
    s2 = new Stack();
}

public void push(int x) {
    s1.push(x);
}

public int pop() {
```

```
    if(s2.size() == 0){
        while(s1.size() > 0) s2.push(s1.pop());
    }

    return s2.pop();
}

public int peek() {
    if(s2.size() == 0){
        while(s1.size() > 0) s2.push(s1.pop());
    }

    return s2.peek();
}

public boolean empty() {
    return s1.size() == 0 && s2.size() == 0;
}
```

## LC-46 全排列

回溯算法

- base case
  - 剪枝，对于无效递归，立即返回
  - 答案，记录后返回
- do choose
  - 将选择移出选择列表
  - 将选择加入结果列表
- cacel choose
  - 将选择加入选择列表
  - 将选择移出结果列表

```
List<List<Integer>> res;

public List<List<Integer>> permute(int[] nums) {
    res = new ArrayList();
    dfs(new ArrayList(), nums);
    return res;
}

private void dfs(List<Integer> curRes, int[] nums){
    // base case
    if(curRes.size() == nums.length){
        res.add(new ArrayList(curRes));
        return;
    }

    // do choose
    for(int num : nums){
        if(curRes.contains(num)) continue;

        curRes.add(num);
        dfs(curRes, nums);
```

```
        // cacel choose
        curRes.remove(Integer.valueOf(num));
    }
}
```

## LC-53 最大子序和

动态规划，由于后续状态仅与前一个状态有关，可不保存DP表

```
// 动态规划(不压缩状态)
public int maxSubArray(int[] nums) {
    int[] dp = new int[nums.length];
    dp[0] = nums[0];
    int res = nums[0];

    for(int i = 1; i < nums.length; i++){
        dp[i] = Math.max(dp[i - 1] + nums[i], nums[i]);
        res = Math.max(res, dp[i]);
    }

    return res;
}
// 压缩状态
public int maxSubArray(int[] nums) {
    int res = nums[0];
    int sum = 0;

    for(int num : nums){
        if(sum < 0) sum = num;
        else sum = sum + num;
        res = Math.max(res, sum);
    }

    return res;
}
```

## LC-8 字符串转换整数

```
public int myAtoi(String s) {
    int res = 0;
    int sign = 1;
    int index = 0;

    // 去除空格
    while(index < s.length() && s.charAt(index) == ' ') index++;

    // 极端情况"  "
    if(index == s.length()) return 0;

    // 明确符号
    if(s.charAt(index) == '-'){
        sign = -1;
        index++;
    }else if(s.charAt(index) == '+'){
        index++;
    }
```

```
    while(index < s.length()){
        char c = s.charAt(index);
        if(c < '0' || c > '9') break;  // 判断数字
        // 判断是否溢出
        if(res != ((res * 10 + (int)c - (int)'0') / 10 )) return sign == 1 ?
Integer.MAX_VALUE : Integer.MIN_VALUE;
        res = res * 10 + (int)c - (int)'0';
        index++;
    }

    return res * sign;
}
```

## LC-124 二叉树中的最大路径和

```
int ans = Integer.MIN_VALUE;

public int maxPathSum(TreeNode root) {
    if (root == null) return 0;
    dfs(root);
    return ans;
}

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

    int left = Math.max(0, dfs(root.left));
    int right = Math.max(0, dfs(root.right));

    ans = Math.max(ans, left + right + root.val);

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

## LC-143 重排链表

- 列表 + 双指针
- 平分链表(快慢指针) + 逆序 + 连接链表

```
// 列表 + 双指针
public void reorderList(ListNode head) {
    if(head == null) return;

    // 存储在列表
    List<ListNode> list = new ArrayList();
    while(head != null){
        list.add(head);
        head = head.next;
    }

    // 双指针连接
    int l = 0;
    int r = list.size() - 1;
    while(l < r){
        list.get(l++).next = list.get(r);
```

```
            if(l == r) break;  // 提前相遇则停止
            list.get(r--).next = list.get(l);
        }
        list.get(l).next = null;  // 处理最后一个节点
    }

    // 平分链表(快慢指针) + 逆序 + 连接链表
    public void reorderList(ListNode head) {
        if(head == null) return;

        // 快慢指针平分链表
        ListNode slow = head;
        ListNode fast = head.next;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode newHead = slow.next;  // 奇偶节点均可适配
        slow.next = null;

        // 逆序
        newHead = reverseList(newHead);

        // 连接链表
        ListNode temp;
        while(head != null && newHead != null){
            temp = head.next;
            head.next = newHead;
            head = temp;

            temp = newHead.next;
            newHead.next = head;
            newHead = temp;
        }

    }

    // 翻转链表
    private ListNode reverseList(ListNode head){
        if(head == null || head.next == null) return head;

        ListNode pre = null;
        ListNode cur = head;
        while(cur != null){
            ListNode temp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = temp;
        }
        return pre;
    }
```

## LC-300 最长递增子序列

- 回溯算法(DFS)，超时
- 动态规划，nums[i]与所有前面的数比较，若大于某个nums[j]，则dp[i] = Math.max(dp[i], dp[j] + 1)

```
// 回溯算法 超时
int res = 0;

public int lengthOfLIS(int[] nums) {
    if(nums.length <= 1) return nums.length;
    List<Integer> curRes = new ArrayList();
    dfs(curRes, nums, 0);
    return res;
}

private void dfs(List<Integer> curRes, int[] nums, int index){
    // base case
    if(index >= nums.length){
        res1 = Math.max(res, curRes.size());
        return;
    }

    for(int i = index; i < nums.length; i++){
        if(curRes.size() > 0 && nums[i] <= curRes.get(curRes.size() - 1)){
            dfs(curRes, nums, i + 1);  // no choose
            continue;
        }else{
            curRes.add(nums[i]);  // do choose
            dfs(curRes, nums, i + 1);
            curRes.remove(curRes.size() - 1);  // cacel choose
        }
    }
    return;
}

// 动态规划
public int lengthOfLIS(int[] nums) {
    if(nums.length <= 1) return nums.length;

    int[] dp = new int[nums.length];
    int res = 0;

    for(int i = 0; i < nums.length; i++){
        dp[i] = 1;
        for(int j = i - 1; j >= 0; j--){
            if(nums[i] > nums[j]) dp[i] = Math.max(dp[i], dp[j] + 1);
        }
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

## LC-470 用Rand7()实现Rand10()

基于(RandX() - 1) * Y + RandY()可以等概率生成[1, X * Y]范围的随机数

- RandX() 等概率生成[1, X]
- RandX() - 1 等概率生成[0, X - 1]
- (RandX() - 1) * Y 等概率生成{0, Y, 2Y, 3Y,...(X-1)Y}
- (RandX() - 1) * Y + RandY()，填补间隙，等概率生成[1, X* Y]

```
public int rand10() {
    while(true){
        int num = (rand7() - 1) * 7 + rand7();  // [1, 49]
        if(num <= 40) return num % 10 + 1;

        int rand9 = num - 40;  // [1, 9]
        num = (rand9 - 1) * 7 + rand7();  //[1, 63]
        if(num <= 60) return num % 10 + 1;

        int rand3 = num - 60;  // [1, 3]
        num = (rand3 - 1) * 7 + rand7();  //[1, 21]
        if(num <= 20) return num % 10 + 1;
    }
}
```

## LC-200 岛屿数量

```
int res;

public int numIslands(char[][] grid) {
    res = 0;

    for(int i = 0; i < grid.length; i++){
        for(int j =0; j < grid[0].length; j++){
            if(grid[i][j] == '1'){
                res++;
                dfs(grid, i, j);
            }
        }
    }

    return res;
}

private void dfs(char[][] grid, int i, int j){
    if(i < 0 || i > grid.length - 1 || j < 0 || j > grid[0].length - 1) return;

    if(grid[i][j] != '1') return;

    grid[i][j] = '2';

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

## LC-82 删除排序链表中的重复元素2

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode vHead = new ListNode(0, head);

    ListNode cur = vHead;

    // 1(cur) 2(i) 2(i) (3)
    while(cur.next != null && cur.next.next != null){
        if(cur.next.val == cur.next.next.val){
            ListNode i = cur.next;
            while(i.next != null && i.val == i.next.val) i = i.next;
            cur.next = i.next;
        }else{
            cur = cur.next;
        }
    }

    return vHead.next;
}
```

## LC-221 最大正方形

- 暴力法，遍历计算
- 动态规划，类似LC-72编辑距离，右下角的矩阵的最大面积右左、上、左上的矩阵最大面积决定
    - 若matrix[i][j]为1，则dp[i][j] = Math. min(dp[i-1][j], dp[i-1][j-1], dp[i][j-1]) + 1

```
// 暴力法
public int maximalSquare(char[][] matrix) {
    int res = 0;
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;

    int rows = matrix.length;
    int column = matrix[0].length;

    for(int i = 0; i < rows; i++){
        for(int j = 0; j < column; j++){
            if(matrix[i][j] == '0') continue;

            res = Math.max(res, 1);

            int curMax = Math.min(rows - i - 1, column - j - 1);
            for(int m = 1; m <= curMax; m++){
                boolean flag = true;
                for(int n = 0; n <= m; n++){
                    if(matrix[i + n][j + m] != '1' || matrix[i + m][j + n] !=
'1') flag = false;
                }
                if(flag == false) break;
                else res = Math.max(res, 1 + m);
            }
        }
    }

    return res * res;
}
```

```java
// 动态规划
public int maximalSquare(char[][] matrix) {
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;

    int rows = matrix.length;
    int column = matrix[0].length;

    int res = 0;
    int[][] dp = new int[rows][column];

    // init
    for(int i = 0; i < rows; i++){
        if(matrix[i][0] == '1'){
            dp[i][0] = 1;
            res = 1;
        }
        else dp[i][0] = 0;
    }
    for(int j = 0; j < column; j++){
        if(matrix[0][j] == '1'){
            dp[0][j] = 1;
            res = 1;
        }
        else dp[0][j] = 0;
    }

    // calculate
    for(int i = 1; i < rows; i++){
        for(int j = 1; j < column; j++){
            if(matrix[i][j] == '1') dp[i][j] = Math.min(dp[i-1][j],
Math.min(dp[i][j-1], dp[i-1][j-1])) + 1;
            else dp[i][j] = 0;
            res = Math.max(res, dp[i][j]);
        }
    }

    return res * res;
}
```

## LC-227 基本计算器2

采用栈存储，保存先前符号

- 数字，执行累加
- 符号或最后一位，执行相应计算

```java
public int calculate(String s) {
    Stack<Integer> stack = new Stack();
    char preOpt = '+';
    int num = 0;
    for(int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        // 执行累加：数字
        if(c <= '9' && c >= '0'){
            num = num * 10 + c - '0';
        }
```

```
        // 执行计算：符号 或 最后一位
        if (c == '+' || c == '-' || c == '*' || c == '/' || i == s.length() - 1)
{
            switch(preOpt){
                case '+' :
                    stack.push(num);
                    break;
                case '-' :
                    stack.push(-num);
                    break;
                case '*' :
                    stack.push(stack.pop() * num);
                    break;
                case '/' :
                    stack.push(stack.pop() / num);
                    break;
            }
            preOpt = c;
            num = 0;
        }
    }

    int res = 0;
    while(stack.size() > 0) res = res + stack.pop();
    return res;
}
```

## LC-48 旋转图像

- 直接旋转
- 上下交换+对角线交换

```
// 直接旋转
public void rotate(int[][] matrix) {
    int size = matrix.length;
    for(int i = 0; i < size / 2; i++){
        for(int j = i; j < size - i - 1; j++){
            int temp = matrix[i][j];
            int iMax = size - 1 - i;
            int jMax = size - 1 - j;
            // 可画图理解
            matrix[i][j] = matrix[jMax][i];
            matrix[jMax][i] = matrix[iMax][jMax];
            matrix[iMax][jMax] = matrix[j][iMax];
            matrix[j][iMax] = temp;
        }
    }
}

// 上下交换+对角线交换
public void rotate(int[][] matrix) {
    int size = matrix.length;

    // 上下交换
    for(int i = 0; i < size / 2; i++){
        int[] temp = matrix[i];
        matrix[i] = matrix[size - 1 - i];
```

```
            matrix[size - 1 - i] = temp;
        }

        // 对角线交换
        for(int i = 0; i < size; i++){
            for(int j = i + 1; j < size; j++){
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }
    }
```

## LC-113 路经总和2

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

public List<List<Integer>> pathSum(TreeNode root, int targetSum) {
    if(root == null) return res;
    List<Integer> path = new ArrayList();
    int sum = 0;
    dfs(root, targetSum, path, sum);
    return res;
}

private void dfs(TreeNode root, int targetSum, List<Integer> path, int sum){
    // 叶子节点
    if(root.left == null && root.right == null){
        if(targetSum == sum + root.val) {
            path.add(root.val);
            res.add(new ArrayList(path));
            path.remove(path.size() - 1);
        }
        return;
    }

    // 非叶子节点
    path.add(root.val);
    sum = sum + root.val;

    if(root.left != null){
        dfs(root.left, targetSum, path, sum);
    }
    if(root.right != null){
        dfs(root.right, targetSum, path, sum);
    }

    // 取消选择
    path.remove(path.size() - 1);
    sum = sum - root.val;
}
```

## LC-1353 最多可以参加的会议数目

基于贪心算法，每次参与结束时间最小的会议

```
public int maxEvents(int[][] events) {
    // 按照开始时间排序
    Arrays.sort(events, (i, j) -> i[0] - j[0]);

    // 小顶堆 存储结束时间
    PriorityQueue<Integer> pq = new PriorityQueue<>();

    int res = 0;
    int curTime = 1;  // 当前时间

    // 遍历到的会议索引
    int i = 0;
    int n = events.length;
    while (i < n || !pq.isEmpty()) {
        // 加入会议 将当前时间开始的会议 按照结束时间放入小顶堆
        while (i < n && events[i][0] == curTime) {
            pq.offer(events[i][1]);
            i++;
        }

        // 去除会议 小顶堆中过期的会议去除掉
        while (!pq.isEmpty() && pq.peek() < curTime) {
            pq.poll();
        }

        // 参加会议 参与结束时间最小的会议
        if (!pq.isEmpty()) {
            pq.poll();
            res++;
        }

        curTime++;
    }

    return res;
}
```

## LC-138 复制带随机指针的链表

- 双指针法，复制，赋值random，拆分
- 哈希存储法，复制，存入Map，赋值next和random

```
// 双指针法
public Node copyRandomList(Node head) {
    if(head == null) return null;

    // 1.复制节点放在后面 a->a->b->b..
    Node cur = head;
    while(cur != null){
        Node temp = new Node(cur.val);
        temp.next = cur.next;
        cur.next = temp;
        cur = temp.next;
```

```
    }

    // 2.赋值新节点的random指针
    cur = head;
    while(cur != null){
        if(cur.random != null) cur.next.random = cur.random.next;
        cur = cur.next.next;
    }

    // 3.分离新旧链表
    cur = head;
    Node newHead = cur.next;
    while(cur.next.next != null){
        Node temp = cur.next;
        cur.next = cur.next.next;
        cur = temp;
    }
    cur.next = null;

    return newHead;
}

// 哈希存储法
public Node copyRandomList(Node head) {
    if(head == null) return null;

    Node cur = head;
    Map<Node, Node> m = new HashMap();

    // 将节点放入Map
    while(cur != null){
        m.put(cur, new Node(cur.val));
        cur = cur.next;
    }

    // 赋值新节点的next和random指针
    cur = head;
    while(cur != null){
        m.get(cur).next = m.get(cur.next);
        if(cur.random != null) m.get(cur).random = m.get(cur.random);
        cur = cur.next;
    }

    return m.get(head);
}
```

## LC-974 和可被K整除的子数组

前缀和，连续子数组

同余定理，若a%b等于c%b，则(a-c)%b为0，即(a-c)能被b整除

分析题目

- 对于[i, j]索引的子数组，其和可通过前缀和数组preSum[j+1] - preSum[i]表示
- 若子数组的和能被K整除，即(preSum[j+1] - preSum[i])%K == 0，则preSum[j+1] % K == preSum[i] % K

- 原问题转换成，求前缀和数组中除K取余相同的索引的组数
    - 由于前缀和可能为负数，则preSum%K为负数则需化为(preSum%K+K)%K

实现方法

- 前缀和+哈希表(逐一统计)
- 前缀和+数组(排列批量统计)，推荐
    - 排列公式，C i,2 = i*(i-1)/2

优化，由于每次将结果统计到哈希表或数组中，可无需记录所有前缀和

```
// 前缀和+哈希表
public int subarraysDivByK(int[] nums, int k) {
    int res = 0;
    int[] preSum = new int[nums.length + 1];
    Map<Integer, Integer> m = new HashMap();

    preSum[0] = 0;
    m.put(0, 1);

    for(int i = 0; i < nums.length; i++){
        preSum[i + 1] = preSum[i] + nums[i];  // 前缀和
        int remainder = (preSum[i + 1] % k + k) % k;  // 前缀和余数(转正)
        // 获取相同前缀和余数的数量 累加到结果
        int curPreSumCount = m.getOrDefault(remainder, 0);
        res = res + curPreSumCount;
        m.put(remainder, curPreSumCount + 1);
    }

    return res;
}

// 前缀和+数组
public int subarraysDivByK(int[] nums, int k) {
    int res = 0;
    int[] preSum = new int[nums.length + 1];
    int[] remainder = new int[k];  // 前缀和余数数组

    preSum[0] = 0;
    remainder[0] = 1;

    for(int i = 0; i < nums.length; i++){
        preSum[i + 1] = preSum[i] + nums[i];  // 前缀和
        remainder[(preSum[i + 1] % k + k) % k]++;  // 前缀和余数(转正)++
    }

    for(int r : remainder){
        if(r != 0) res = res + (r * (r - 1) / 2);  // 排列公式计算
    }

    return res;
}

// 优化 不保留前缀和
public int subarraysDivByK(int[] nums, int k) {
    int res = 0;
    int preSum = 0;
```

```
    int[] remainder = new int[k];

    remainder[0] = 1;

    for(int i = 0; i < nums.length; i++){
        preSum = preSum + nums[i];
        remainder[(preSum % k + k) % k]++;
    }

    for(int r : remainder){
        if(r != 0) res = res + (r * (r - 1) / 2);
    }

    return res;
}
```

## LC-662 二叉树的最大宽度

由于空节点存在，需要通过二叉树节点的下标索引来计算节点之间的距离

层序遍历(BFS)+节点索引计算

- 左子节点索引 = 节点索引*2 + 1
- 右子节点索引 = 节点索引*2 + 2

```
class TreeNodeWithIndex {  // 带索引的节点对象
    TreeNode tn;
    int index;
    TreeNodeWithIndex(TreeNode tn, int index){
        this.tn = tn;
        this.index = index;
    }
}
public int widthOfBinaryTree(TreeNode root) {
    if(root == null) return 0;

    Queue<TreeNodeWithIndex> q = new LinkedList();
    q.offer(new TreeNodeWithIndex(root, 0));
    int res = 1;

    while(q.size() > 0){
        int size = q.size();
        int leftIndex = 0;  // 左节点索引
        int rightIndex = 0;  // 右节点索引
        for(int i = 0; i < size; i++){
            TreeNodeWithIndex node = q.poll();
            if(i == 0) leftIndex = node.index;  // 左节点
            if(i == size - 1) rightIndex = node.index;  // 右节点
            if(node.tn.left != null) q.offer(new TreeNodeWithIndex(node.tn.left,
node.index * 2 + 1));  // 下一层左子节点
            if(node.tn.right != null) q.offer(new
TreeNodeWithIndex(node.tn.right, node.index * 2 + 2));  // 下一层右子节点
        }
        res = Math.max(res, rightIndex - leftIndex + 1);  // 当前层的宽度
    }

    return res;
```

```
}
```

## LC-110 平衡二叉树

```
boolean flag = true;

public boolean isBalanced(TreeNode root) {
    if(root == null) return true;
    dfs(root, 0);
    return flag;
}

private int dfs(TreeNode root, int depth){
    if(root == null) return depth - 1;

    int left = dfs(root.left, depth + 1);
    int right = dfs(root.right, depth + 1);

    if(Math.abs(left - right) > 1) flag = false;

    return Math.max(left, right);
}
```

## LC-198 打家劫舍

动态规划，对于第i家dp[i]，偷则dp[i-2]+nums[i]，不偷则dp[i-1]，可不保留dp数组优化

```
public int rob(int[] nums) {
    if(nums.length == 1) return nums[0];
    if(nums.length == 2) return Math.max(nums[0], nums[1]);

    int first = nums[0];
    int second = Math.max(nums[0], nums[1]);

    for(int i = 2; i < nums.length; i++){
        int curRes = Math.max(first + nums[i], second);
        first = second;
        second = curRes;
    }

    return second;
}
```

## LC-213 打家劫舍2

对于第一家和最后一家，只有一家可以偷，因此化为两次dp取最大值

```
public int rob(int[] nums) {
    if(nums.length == 1) return nums[0];
    if(nums.length == 2) return Math.max(nums[0], nums[1]);

    // [0, nums.length - 2]
    int first = nums[0];
    int second = Math.max(nums[0], nums[1]);

    for(int i = 2; i < nums.length - 1; i++){
```

```
            int curRes = Math.max(first + nums[i], second);
            first = second;
            second = curRes;
        }

        int res = second;

        // [1, nums.length - 1]
        first = nums[1];
        second = Math.max(nums[1], nums[2]);

        for(int i = 3; i < nums.length; i++){
            int curRes = Math.max(first + nums[i], second);
            first = second;
            second = curRes;
        }

        res = Math.max(res, second);

        return res;
    }
```

### LC-287 寻找重复数

索引->值->索引，会形成环，类似找到环状链表的入口

```
public int findDuplicate(int[] nums) {
    int slow = nums[0];
    int fast = nums[nums[0]];

    while(slow != fast){
        slow = nums[slow];
        fast = nums[nums[fast]];
    }

    fast = 0;
    while(slow != fast){
        slow = nums[slow];
        fast = nums[fast];
    }

    return slow;
}
```

### 面试题 02.04. 分割链表

```
public ListNode partition(ListNode head, int x) {
    ListNode sHead = new ListNode();
    ListNode bHead = new ListNode();

    ListNode cur = head;
    ListNode sHeadCur = sHead;
    ListNode bHeadCur = bHead;
    while(cur != null){
        if(cur.val < x){
            sHeadCur.next = cur;
```

```
            sHeadCur = cur;
            cur = cur.next;
        }else{
            bHeadCur.next = cur;
            bHeadCur = cur;
            cur = cur.next;
        }
    }

    sHeadCur.next = bHead.next;
    bHeadCur.next = null;

    return sHead.next;
}
```

### 课程表

拓扑排序算法

- 入度列表，0表示无依赖
- 邻接矩阵，存储依赖当前节点的节点
- 队列，用于BFS确定遍历节点顺序

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    if(numCourses <= 0) return false;
    if(prerequisites.length == 0) return true;

    int[] indegrees = new int[numCourses];
    List<List<Integer>> adj= new ArrayList();

    // init adj
    for(int i = 0; i < numCourses; i++){
        adj.add(new ArrayList());
    }

    // computer indegrees adj
    for(int[] prerequisite : prerequisites){
        int first = prerequisite[1];
        int second = prerequisite[0];
        indegrees[second]++;
        adj.get(first).add(second);
    }

    // BFS
    Queue<Integer> q = new LinkedList();
    int courseNum = 0;

    for(int i = 0; i < numCourses; i++){
        if(indegrees[i] == 0) q.offer(i);
    }

    while(q.size() > 0){
        int course = q.poll();
        courseNum++;
        for(int relateCourse : adj.get(course)){
            indegrees[relateCourse]--;
            if(indegrees[relateCourse] == 0) q.offer(relateCourse);
```

```
            }
        }

        return courseNum == numCourses;
    }
```