异位词 -- 两个字符串之间的字符出现频率一致，但顺序不一致

暴力法:
sort 两个字符串, 比较两个字符串是否相等
HashMap:
1. int[26], 初始化26个字母的数组做字母表
2. 思路看 HashMap 那种, 只是用了 Array 代替 HashMap, array index 对应字母
3.遍历 Array 看有没有非 0 元素, 有则代表不是异位词

关键 **key = s.charAt(i) - 'a'**

异位词分组

核心是利用 Map
遍历数组
1. 将字符串转 toCharArray
2. keyString = Arrays.sort(charArray)
3. 查 Map 里是否有 keyString, 没有则新增当前 keyString 到 Map, value 为 新建 List
4. 查 keyString 在 Map, 查到后添加

Hash

特性, 访问 O(1), 所以可用于高频查询数据的场景，来加速访问速度

Two Sum

核心公式
map.containsKey(target - nums[i])

Stack

valid parentheses

单 stack, 遇到 左括号 则对应的右括号入栈
遇到 右括号，如果 栈为空 或 栈顶出栈元素 不是当前右括号 则表示 此字符串为不合法的括号
最后检查 return stack.empty() // expect true

min stack

https://leetcode-cn.com/problems/min-stack/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-38/ 三种解法

新的最小值入栈之前，把当前最小值入栈
出栈的如果是当前最小值，再出一次栈，出栈结果赋值给 min

```
class MinStack {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();

    public void push(int x) {
        if (stack2.empty() || x <= stack2.peek()) {
            stack2.push(x);
        }
        stack1.push(x);
    }

    public void pop() {
        if (stack1.pop() == stack2.peek()) stack2.pop();
    }

    public int top() {
        return stack1.peek();
    }

    public int getMin() {
        return stack2.peek();
    }
}
```

largest rectangle in histogram

核心: 找左右边界, 边界中间的内容用于依次计算最大面积
1.需要一个堆(堆顶下面的元素是它的左边界), 给堆 push 一个 -1 作为 default left bound
2.遍历 heights
3.(loop)如果当前棒子比堆顶棒子矮, 则说明堆顶棒子的 right bound找到了, 弹出堆顶棒子并计算此棒子的面积(栈顶为left bound, 当前 index 为 right bound)
4.遍历后，检查堆是否只剩下 -1, 如果还有其他棒子,不断弹出堆顶的棒子并计算堆顶面积(棒子弹出后, 此棒子 left bound 为栈顶, right bound 为 heights.length)

trapping rain water

设置左右两边指针
设置 leftmax, rightmax, 默认 0
让左右两边指针相遇

- 如果左指针高度小于右边, 则计算左指针能装多少水(这代表右边比左边高。先更新当前指针跟当前max谁最高, 再用最大高度减去当前高度, 得出装的水数量)
e.g: res += Math.max(leftmax, height[left]) - height[left]

- 否则计算右边指针能装多少水

缩小左或右指针范围 直到相遇

Queue

Sliding Window Maximum

- https://www.youtube.com/watch?v=fbkvdWUS5Ic

1. 一共有 n - k + 1 个窗户
2. 需要一个优先队列存 index, head 是当前窗口最大值, tail是最小值, 如果新入的值比tail大, tail 出列, 反复此过程
3. 当索引到的位置足够组成一个窗口时, push head(largest num) 到结果
4. 窗口每次移动时, 检查当前 head 所存的 index 是否在窗口, 不在了则出列

```
function slidingWindow模板(nums) {
    var left = 0, right = -1;//滑动窗口为s[l...r]
    var n = nums.length
    定义其他中间量
    while (left < n) { //大循环
      //整个循环是从left ==0, right == -1这个空窗口开始
      //到left = n, r = n -1这个空窗口结果
      //扩大窗口
      if (right + 1 < n && 对right + 1进行检测，如果它不满足某些条件，继续往右扩大窗口) {
        right++
        维护中间量
      } else {//将窗口往左移动
        将中间量回退，比如上面加加，这里就减减
        left++ //这里必须放在这里，不能被括号包起玉
      }
      //计算最优值
    }
    return 最优值
}
```

循环双端队列

https://www.youtube.com/watch?v=ia__kyuwGag

环形队列（循环队列）

参考资料

- Java版-数据结构-队列（循环队列）
- Using an Array to represent a Circular Queue

主要概念

- 先进先出
- capacity：数组容量
- front：表示队列队首，始终指向队列中的第一个元素（当队列空时，front指向索引为0的位置）
- tail：表示队列队尾，始终指向队列中的最后一个元素的下一个位置
- 元素入队，维护tail的位置，进行tail++操作，计算公式：(tail + 1) % capacity
- 元素出队，维护front的位置，进行front++操作，计算公式：(front + 1) % capacity
- 在循环队列中，总是浪费一个空间，来区分队列为满时和队列为空时的情况，也就是当 (tail + 1) % capacity == front的时候，表示队列已经满了，当front == tail的时候，表示队列为空。

疑问解答

- 为什么front的计算公式为：(front + 1) % capacity？

1. 元素出队时，弹出当前front的元素，front向数组右边移动一位，所以新的front = front + 1
2. 因为front的取值范围是[0, capacity - 1]，当front = capacity - 1时（数组的最后一位）front + 1就会出现数组越界！这时有2种方法可以处理：
   - 2.1. 判断数组越界后，直接将front赋值为0。因为是环形数组，开始循环了。
   - 2.2. 根据"余数定理"，%操作可以保证结果范围在[0, capacity - 1]内。因为front的最大值是capacity - 1。
3. 取余操作相对于每次判断是否最大索引值的方式，代码更简洁，但相对难理解。