

## 面试题 08.03. 魔术索引

分支+剪叶

```
class Solution {
    public int findMagicIndex(int[] nums) {
        return getAnswer(nums, 0, nums.length - 1);
    }

    public static int getAnswer(int[] nums, int left, int right){
        if(left > right)
            return -1;
        int mid = (left + right) / 2;
        int leftAnswer = getAnswer(nums, left, mid - 1);
        if(leftAnswer != -1){ //如果在左边能够搜索到（必须首先搜索左半部分， 因为即使
mid点符合，也有可能左边仍有最小的）
            return leftAnswer;
        }
        else if(nums[mid] == mid){ //左边不能搜索到则判断mid点
            return mid;
        }
        else{ //左半部分 和 mid点都不符合 则搜索右半部分
            return getAnswer(nums, mid + 1, right);
        }
    }
}
```

## 剑指 Offer 39. 数组中出现次数超过一半的数字

子烁

```
class Solution {
    /*
        必然会存在这么一个元素，而且只有一个，且超过一半，所以这样遍历一定会遍历到这一个
    */
    public int majorityElement(int[] nums) {
        int flag = nums[0], times = 1;
        for(int i = 1; i < nums.length; i++){
            if(times == 0){
                flag = nums[i];
                times = 1;
            }
            else{
                if(nums[i] == flag)
                    times++;
                else
                    times--;
            }
        }
    }
}
```

```
        return flag;
    }
}
```

## 剑指 Offer 40. 最小的k个数

子烁

```
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if(arr.length < k || k < 1)
            return new int[k];
        PriorityQueue<Integer> queue = new PriorityQueue<>((o1, o2) -> o2 - o1);
        PriorityQueue<Integer> queue = new PriorityQueue<>(new Comparator<Integer>
() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });

        for(int num : arr){
            if(queue.size() < k){
                queue.add(num);
                continue;
            }
            if(queue.peek() > num){ //如果小于堆中的最大值，则放入堆中
                queue.poll();
                queue.add(num);
            }
        }
        int[] res = new int[k];
        int n = 0;
        for(int num : queue){
            res[n] = num;
            n++;
        }
        return res;
    }
}
```

## 剑指 Offer 45. 把数组排成最小的数

子烁

```
public class Test {
    public static void main(String[] args) {
        int[] nums = {3, 30, 34, 5, 9};
```

```

    }

    public static String Solution(int[] nums) {
        String[] arr = new String[nums.length];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = String.valueOf(nums[i]);
        }
        Arrays.sort(arr, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return (o1 + o2).compareTo(o2 + o1);
            }
        });
        StringBuilder builder = new StringBuilder();
        for (String s : arr) {
            builder.append(s);
        }
        return builder.toString();
    }
}

```

## 剑指 Offer 49. 丑数

子烁

```

class Solution {
    public int nthUglyNumber(int n) {
        int[] uglyNumbers = new int[n];
        uglyNumbers[0] = 1;
        int p2 = 0, p3 = 0, p5 = 0;
        for(int i = 1; i < n; i++){
            int lastUglyNumber = uglyNumbers[i - 1];
            while(lastUglyNumber >= uglyNumbers[p2] * 2){
                p2++;
            }
            while(lastUglyNumber >= uglyNumbers[p3] * 3){
                p3++;
            }
            while(lastUglyNumber >= uglyNumbers[p5] * 5){
                p5++;
            }
            uglyNumbers[i] = Math.min(uglyNumbers[p2] * 2,
Math.min(uglyNumbers[p3] * 3, uglyNumbers[p5] * 5));
        }
        return uglyNumbers[n - 1];
    }
}

```

//n过大时，会超出时间限制

```

class Solution {
    public int nthUglyNumber(int n) {
        if(n == 1)
            return 1;
        int n1 = 1, m = 2;
        while(true){
            if(isUglyNumber(m)){
                n1++;
                if(n1 == n)
                    break;
                m++;
            }
            else{
                m++;
            }
        }
        return m;
    }
    public boolean isUglyNumber(int num){
        while(num % 2 == 0){
            num /= 2;
        }
        while(num % 3 == 0){
            num /= 3;
        }
        while(num % 5 == 0){
            num /= 5;
        }
        return num == 1;
    }
}

```

## 【位运算】剑指 Offer 56 - I. 数组中数字出现的次数

解答

```

class Solution {
    public int[] singleNumbers(int[] nums) {
        int c = 0;
        for(int num : nums){    //找出所有的异或结果，此结果为a,b的异或，肯定不为0，否则两者相等
            c ^= num;
        }

        int mark = 1;    //c的二进制编码中肯定存在某一位1，则设置为mark的二进制中的那一位为1，其它为0
        while((c & mark) == 0){
            mark <<= 1;
        }
    }
}

```

```

int a = 0, b = 0;
for(int num : nums){
    if((num & mark) == 0){ //与mark异或为0的则必然是在那一位为0的一组中
        a ^= num;
    }else{                //与mark异或为1的则必然是在那一位为1的另外一组中
        b ^= num;
    }
}
int[] res = {a,b};
return res;
}
/*
    关键点在分组上: 4 ->100 ; 1 ->001; 6 ->110; 4 ->100 ; 所有数字异或值为 k (这里是111), 只要找到一个k中, 位数为1的任意位号mask (这里用的是 while((k & mask) == 0) { mask <= 1; } ), 显然第一位就是mask, mask这个位号表示的是那两个不重复数的二进制 (001、110) 在这个位号上不同时为0或1的位置, 那么锁定这样的位置后, 以同样方法 (使用(num & mask) == 0), 将mask位不为1的剔出来为一组, 而另一组中必然会有mask位为1的数, 这样就实现了不重复的两个数分到了不同组, 而那些重复数必然被分到相同的组中, 最终被抵消。
*/
}

```

## 剑指 Offer 53 - I. 在排序数组中查找数字 I

子烁

```

class Solution {
    public int search(int[] nums, int target) {
        int index = Arrays.binarySearch(nums, target); //二分查找位置
        if(index >= nums.length || index < 0)
            return 0;
        int n = 1, left = index - 1, right = index + 1;
        while(left >= 0 && nums[left--] == target) //向左边搜索
            n++;
        while(right < nums.length && nums[right++] == target) //向右边搜索
            n++;
        return n;
    }
}

```

## 【滑动窗口】剑指 Offer 57 - II. 和为s的连续正数序列

子烁

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。

```

class Solution {
    public int[][] findContinuousSequence(int target) {
        List<List<Integer>> res = getResList(target);
    }
}

```

```

int[][] result = new int[res.size()][];
int n = 0;
for(List<Integer> m : res){
    int[] level = new int[m.size()];
    for(int i = 0; i < m.size(); i++){
        level[i] = m.get(i);
    }
    result[n] = level;
    n++;
}
return result;
}

private List<List<Integer>> getResList(int sum){
    List<List<Integer>> result = new ArrayList<>();
    int left = 1, right = 2, currSum = left + right;
    while(left < right && right < sum){
        //如果当前值大于目标值，则左移窗口以减小当前值
        while(currSum > sum && left < sum / 2){ //这里不用判断left < right，因为即使这样，最终left < sum / 2也会退出循环，而且此时就算不满足left < right，那么运行到外训最后一位时候，也会判定停止循环
            currSum -= left;
            left++;
        }
        if(currSum == sum){
            result.add(getFence(left, right));
        }
        //如果当前值小于目标值，则右移动窗口以增加当前值
        right++;
        currSum += right;
    }
    return result;
}

private List<Integer> getFence(int a, int b){
    List<Integer> res = new ArrayList<>();
    for(int i = a; i <= b; i++){
        res.add(i);
    }
    return res;
}
}

```

## 【滑动窗口】剑指 Offer 57. 和为s的两个数字

子烁

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们的和正好是s。如果有多对数字的和等于s，则输出任意一对即可。

```

class Solution {
    public int[] twoSum(int[] nums, int target) {
        Arrays.sort(nums);
        int left = 0, right = nums.length - 1;
        while(left < right && right < target){
            if(nums[left] + nums[right] == target){
                int[] res = {nums[left], nums[right]};
                return res;
            }
            if(nums[left] + nums[right] < target){
                left++;
            }
            else{
                right--;
            }
        }
        return new int[2];
    }
}

```

## 剑指 Offer 03. 数组中重复的数字

在一个长度为  $n$  的数组 `nums` 里的所有数字都在  $0 \sim n-1$  的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

```

class Solution {
    public int findRepeatNumber(int[] nums) {
        int[] arr = new int[nums.length + 1];
        for(int num : nums){
            arr[num]++;
        }
        for(int i = 0; i <= arr.length; i++){
            if(arr[i] > 1)
                return i;
        }
        return 0;
    }
}

```

## 剑指 Offer 04. 二维数组中的查找

子烁

在一个  $n * m$  的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```

/*
***** --> (此处为*标的中间值)

```

```

        *
        *
        *
        *
        *
        *
        *
        *
        *

*/

class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0)
            return false;
        int rows = matrix.length, cols = matrix[0].length;
        int x = 0, y = cols - 1;
        while(x < rows && y >= 0){
            if(matrix[x][y] == target)
                return true;
            if(matrix[x][y] > target)
                y--;
            else
                x++;
        }
        return false;
    }
}
```

## 剑指 Offer 11. 旋转数组的最小数字

子烁

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一个旋转，该数组的最小值





```

        b = (double)(area) / (double)(a);
        if((b % 1) == 0){
            break;
        }
        a++;
    }
    int row = a, col = (int)b;
    int[] result = {row, col};
    return result;
}
}

class Solution {
    public int[] constructRectangle(int area) {
        int width = (int) Math.sqrt(area);
        while (area % width != 0) {
            width--;
        }
        return new int[]{area / width, width};
    }
}

```

## 全排列

```

class Solution {
    List<List<Integer>> resultList;
    public List<List<Integer>> permute(int[] nums) {
        resultList = new ArrayList<>();
        List<Integer> arr = new LinkedList<>();
        for(int num : nums){
            arr.add(num);
        }
        helper(arr, new ArrayList<>(), nums.length, 0);

        return resultList;
    }

    private void helper(List<Integer> arr, ArrayList<Integer> res, int length, int index){
        if(index == length){
            resultList.add(new ArrayList<>(res)); //此处必须重新构建new ArrayList<>(res), 不能add(res) 不要问为啥~
            return;
        }

        for(int i = 0; i < arr.size(); i++){
            int num = arr.get(i);
            res.add(num);
            arr.remove(i);
            helper(arr, res, length, index + 1);
            arr.add(i, num);
        }
    }
}

```

```

        res.remove(res.size() - 1);
    }
}

```

## 回溯

```

class Solution {
    List<List<Integer>> res = new LinkedList<>();
    public List<List<Integer>> permute(int[] nums) {
        LinkedList<Integer> track = new LinkedList<>();
        backtrack(nums, track);
        return res;
    }

    // 路径: 记录在 track 中
    // 选择列表: nums 中不存在于 track 的那些元素
    // 结束条件: nums 中的元素全都在 track 中出现
    private void backtrack(int[] nums, LinkedList<Integer> track){
        if(track.size() == nums.length){
            res.add(new LinkedList<>(track));
            return;
        }
        for(int i = 0; i < nums.length; i++){
            // 排除不合法的选择
            if(track.contains(nums[i]))
                continue;
            // 做选择
            track.add(nums[i]);
            // 进入下一层决策树
            backtrack(nums, track);
            // 取消选择
            track.removeLast();
        }
    }
}

```

## 【动态规划】[0-1背包问题]

```

/**
 * Description
 * Author cloudr
 * Date 2020/8/31 18:32
 * Version 1.0
 */
public class bag0_1 {
    public static int solution(int Capacity, int N, int[] weights, int[] values) {
        int[][] dp = new int[N + 1][Capacity + 1]; //dp[i][c]: 在前i件物品、背包的容量为c时, 这种情况下可以装下的最大价值是dp[i][c]。
    }
}

```

```

        for (int i = 1; i <= N; i++) {
            for (int c = 1; c <= Capacity; c++) {
                if (c - weights[i - 1] < 0) {
                    //当前背包容量装不下, 只能选择不装入背包
                    dp[i][c] = dp[i - 1][c];
                    continue;
                }
                dp[i][c] = Math.max(dp[i - 1][c], dp[i - 1][c - weights[i - 1]] +
values[i - 1]);
                //                不放入                放入当前物品
            }
        }
        return dp[N][Capacity];
    }
    public static void main(String[] args) {
        int[] weights = {2, 1, 3};
        int[] values = {4, 2, 3};
        int Capacity = 4;
        int N = 3;
        System.out.println(solution(Capacity, N, weights, values));
    }
}

```

## 【动态规划】面试题 17.13. 恢复空格

子烁

哦，不！你不小心把一个长篇文章中的空格、标点都删掉了，并且大写也弄成了小写。像句子"I reset the computer. It still didn't boot!"已经变成了"iresetthecomputeritstill didntboot"。在处理标点符号和大小写之前，你得先把它断成词语。当然了，你有一本厚厚的词典dictionary，不过，有些词没在词典里。假设文章用sentence表示，设计一个算法，把文章断开，要求未识别的字符最少，返回未识别的字符数。

```

class Solution {
    public int respace(String[] dictionary, String sentence) {
        int[] dp = new int[sentence.length() + 1]; //可已经匹配的单词的最大长度
        dp[0] = 0;
        for(int i = 1; i <= sentence.length(); i++){ //因为substring()左闭右开
            dp[i] = dp[i - 1];
            for(String word : dictionary){
                if(i < word.length())
                    continue;
                String temp = sentence.substring(i - word.length(), i);
                if(!temp.equals(word))
                    continue;
                dp[i] = Math.max(dp[i], dp[i - word.length()] + word.length());
            }
        }
        return sentence.length() - dp[sentence.length()];
    }
}

```

## 【动态规划】322. 零钱兑换

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`。

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1]; //当目标金额为 i 时, 至少需要 dp[i] 枚硬币凑出。
        Arrays.fill(dp, amount + 1); //最大值时不要使用Integer.MAX_VALUE, 否则dp[i - coin] + 1有可能会出现负值
        dp[0] = 0; //总金额为0必不需要硬币
        for(int i = 0; i <= amount; i++){
            for(int coin : coins){
                if(i - coin < 0)
                    continue;
                dp[i] = Math.min(dp[i], dp[i - coin] + 1); //如果不符合, dp[i] 还是为amount + 1
            }
        }
        return (dp[amount] == amount + 1) ? -1 : dp[amount];
    }
}
/*
为啥 dp 数组初始化为 amount + 1 呢, 因为凑成 amount 金额的硬币数最多只可能等于 amount (全用 1 元面值的硬币), 所以初始化为 amount + 1 就相当于初始化为正无穷, 便于后续取最小值。 不能设置为 -1, 否则恒为-1
*/
```

## 【动态规划】983. 最低票价

在一个火车旅行很受欢迎的国度，你提前一年计划了一些火车旅行。在接下来的一年里，你要旅行的日子将以一个名为 `days` 的数组给出。每一项是一个从 1 到 365 的整数。

火车票有三种不同的销售方式：

- 一张为期一天的通行证售价为 `costs[0]` 美元；
- 一张为期七天的通行证售价为 `costs[1]` 美元；
- 一张为期三十天的通行证售价为 `costs[2]` 美元。

通行证允许数天无限制的旅行。例如，如果我们在第 2 天获得一张为期 7 天的通行证，那么我们可以连着旅行 7 天：第 2 天、第 3 天、第 4 天、第 5 天、第 6 天、第 7 天和第 8 天。

返回你想要完成在给定的列表 `days` 中列出的每一天的旅行所需要的最低消费。

```
class Solution {
    public int mincostTickets(int[] days, int[] costs) {
        //判断
        if(days == null || days.length == 0 ||
            costs == null || costs.length == 0) {
            return 0;
        }

        //dp表示到了当天花的最低票价
        int[] dp = new int[days[days.length - 1] + 1];

        //base case: 第0天一定不用买票 则花费0元
        dp[0] = 0;
        //标记一下需要买票的日子
        for(int day: days) {
            dp[day] = Integer.MAX_VALUE;
        }

        for(int i = 1; i < dp.length; i++) {
            //不需要买票
            if(dp[i] == 0) {
                //不需要买票花费的钱就是前一天的花费
                dp[i] = dp[i - 1];
                continue;
            }

            int n1 = dp[i - 1] + costs[0]; //当天需要买票
            /**如果今天距离第一天已经超过7天了
             * 则花费: dp[i-7](7天前已经花费的钱)+cost[1](7天前买了一张7天的票)
             * 否则就是直接第一天买了一张7天票
             */
            int n2 = i > 7 ? dp[i - 7] + costs[1] : costs[1];
            //30天与7天 同理
            int n3 = i > 30 ? dp[i - 30] + costs[2] : costs[2];

            dp[i] = Math.min(n1, Math.min(n2, n3));
        }
        //最后一天花费多少钱
        return dp[days[days.length - 1]];
    }
}
```

作者: eddieVim

链接: <https://leetcode-cn.com/problems/minimum-cost-for-tickets/solution/dong-tai-gui-hua-jie-ti-xiang-xi-zhu-shi-by-eddiev/>

来源: 力扣 (LeetCode)

著作权归作者所有。商业转载请联系作者获得授权, 非商业转载请注明出处。

## 【动态规划】[航班票价最低]

从一个城市飞往另一个城市有很多条航班路线，在做好规划的情况下，为了保证票价最低，允许通过中转的方式来乘坐飞机。假设从A城市到B城市的航班价格为N,给你W个城市的航班信息R,输入任意的A城市和B城市，求最便宜的转机价格M.

W = 3, R = {{0, 1, 600}, {1, 2, 800}, {0, 2, 1300}}, A = 0, B = 2;  
返回: M = 1100

```
public class CVTE_0907 {
    public static void main(String[] args) {
        int[][] R = {{0, 1, 600}, {1, 2, 800}, {0, 2, 1300}};
        int W = 3;
        int A = 0;
        int B = 2;
        System.out.println(findMin(W, R, A, B));
    }

    public static int findMin(int W, int[][] R, int A, int B){
        int[] dp = new int[B - A + 1]; //dp[i] : 到达当前路径时的最小花费
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[0] = 0;
        for(int i = A + 1; i <= B; i++){
            for(int[] trace : R){
                int from = trace[0];
                int to = trace[1];
                int cost = trace[2];
                if(i == to && from >= A){
                    dp[i] = Math.min(dp[i], dp[i - (to - from)] + cost);
                }
            }
        }
        return dp[B - A];
    }
}
```

## 【动态规划】121. 买卖股票的最佳时机

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一次），设计一个算法来计算你能获取的最大利润。

注意：你不能在买入股票前卖出股票。

```
class Solution {
    public int maxProfit(int[] prices) {
        if(prices.length == 0)
            return 0;
        int n = prices.length;
```

```

int[][] dp = new int[n][2];
dp[0][0] = 0;
dp[0][1] = -prices[0];
for(int i = 0; i < n; i++){ //或者直接从1开始，去掉下面的continue
    if(i == 0)
        continue;
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);    //未持有
    max{昨天也未持有， 今天买入}
    dp[i][1] = Math.max(dp[i - 1][1], 0 - prices[i]);            //持有
    max{昨天也持有， 今天卖掉}
    // dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
}
return dp[n - 1][0];
}
}

```

```

class Solution {
    public int maxProfit(int[] prices) {
        if(prices.length == 0)
            return 0;
        int n = prices.length;
        int dp_0 = 0, dp_1 = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            dp_0 = Math.max(dp_0, dp_1 + prices[i]);
            // dp_1 = Math.max(dp_1, dp_0 - prices[i]); dp_0 始终为0，而且如果装出现
            连续几个dp_0不为空，则错误
            dp_1 = Math.max(dp_1, - prices[i]);
        }
        return dp_0;
    }
}

```

## 【动态规划】122. 买卖股票的最佳时机 II

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）。

### 贪心算法

```

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

```



```
}
```

## DP

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int dp_0 = 0, dp_1 = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            int temp = dp_0;
            dp_0 = Math.max(dp_0, dp_1 + prices[i]);
            dp_1 = Math.max(dp_1, dp_0 - prices[i]);
        }
        return dp_0;
    }
}
```

## 【动态规划】123. 买卖股票的最佳时机 III

```
class Solution {
    public int maxProfit(int[] prices) {
        int len = prices.length;
        if (len < 2) { //为1也不行, 否则只能买入不能卖出
            return 0;
        }

        // 为了使得第 2 维的数值 1 和 2 有意义, 这里将第 2 维的长度设置为 3
        int[][][] dp = new int[len][3][2];

        // 理解如下初始化
        // 第 3 维规定了必须持股, 因此是 -prices[0]
        dp[0][1][1] = -prices[0];
        // 还没发生的交易, 持股的时候应该初始化为负无穷
        dp[0][2][1] = Integer.MIN_VALUE;

        for (int i = 1; i < len; i++) {
            // 转移顺序先持股, 再卖出
            dp[i][1][1] = Math.max(dp[i - 1][1][1], -prices[i]);
            dp[i][1][0] = Math.max(dp[i - 1][1][0], dp[i - 1][1][1] + prices[i]);
            dp[i][2][1] = Math.max(dp[i - 1][2][1], dp[i - 1][1][0] - prices[i]);
            //dp[i - 1][1][0]
            dp[i][2][0] = Math.max(dp[i - 1][2][0], dp[i - 1][2][1] + prices[i]);
        }
        return Math.max(dp[len - 1][1][0], dp[len - 1][2][0]);
    }
}
```

```

class Solution {
    public int maxProfit(int[] prices) {
        int len = prices.length;
        if (len < 2) {
            return 0;
        }

        int dp_10 = 0, dp_11 = Integer.MIN_VALUE;
        int dp_20 = 0, dp_21 = Integer.MIN_VALUE;
        for(int i = 0; i < len; i++){
            dp_10 = Math.max(dp_10, dp_11 + prices[i]);
            dp_11 = Math.max(dp_11, - prices[i]);
            dp_20 = Math.max(dp_20, dp_21 + prices[i]);
            dp_21 = Math.max(dp_21, dp_10 - prices[i]);
        }

        return Math.max(dp_10, dp_20);
    }
}

```

## 【动态规划】188. 买卖股票的最佳时机 IV

给定一个数组，它的第  $i$  个元素是一支给定的股票在第  $i$  天的价格。

设计一个算法来计算你能获取的最大利润。你最多可以完成  $k$  笔交易。

```

class Solution {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;
        if(n < 2)
            return 0;
        if(k >= n / 2) //每天交易一次，n天之多交易n/2此，超过则可认为每一天都可以交易
            return maxProfit_k_inf_1(prices);

        int[][][] dp = new int[n][k + 1][2];
        for(int i = 0; i < n; i++){
            for(int j = 1; j <= k; j++){
                if(i == 0){
                    dp[i][j][0] = 0;
                    dp[i][j][1] = -prices[0];
                    continue;
                }
                dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j][1] +
prices[i]);
                dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] -
prices[i]);
            }
        }
        int a = 1, res = 0;
        while(a <= k ){
            if(dp[n - 1][a][0] > res)

```

```

        res = dp[n - 1][a][0];
        a++;
    }
    return res;
}

int maxProfit_k_inf_1(int[] prices) {
    int n = prices.length, sum = 0;
    for(int i = 1; i < n; i++){
        if(prices[i] > prices[i - 1])
            sum += prices[i] - prices[i - 1];
    }
    return sum;
}

int maxProfit_k_inf(int[] prices) {
    if(prices.length < 0)
        return 0;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for(int i = 0; i < prices.length; i++){
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, dp_i_0 - prices[i]);
    }
    return dp_i_0;
}
}

```

## 【动态规划】LCP 07. 传递信息

```

class Solution {
    public int numWays(int n, int[][] relation, int k) {
        /*
        dp[i][j] 表示数组的第 i 轮传递给编号 j 的人的方案数
        若能传递给编号 y 玩家的所有玩家编号 x1,x2,x3... , 则第 i+1 轮传递信息给编号 y
        玩家的递推方程为
        dp[i+1][y] = sum(dp[i][x1],dp[i][x2],dp[i][x3]...),
        其递推形式即
        dp[i+1][y] += dp[i][x]
        */
        int[][] dp = new int[k + 1][n];
        dp[0][0] = 1; // 第0轮到达编号零, 方案数为1
        for(int a = 1; a <= k; a++){
            for(int b = 0; b < relation.length;b++){
                dp[a][relation[b][1]] += dp[a - 1][relation[b][0]]; //relation[b]
[0]传递给relation[b][1]
            }
        }
        return dp[k][n - 1];
    }
}

```

## 面试题 10.01. 合并排序的数组

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。

```
class Solution {
    public void merge(int[] A, int m, int[] B, int n) {
        /*
        int j = 0;
        for(int i = m; i < A.length; i++){
            A[i] = B[j];
            j++;
        }
        Arrays.sort(A);
        */
        int[] arr = new int[m + n];
        int a = 0, b = 0, k = 0;
        while(a < m && b < n){
            if(A[a] <= B[b])
                arr[k++] = A[a++];
            else
                arr[k++] = B[b++];
        }
        while(a < m){
            arr[k++] = A[a++];
        }
        while(b < n){
            arr[k++] = B[b++];
        }
        int c = 0;
        for(int num : arr){
            A[c] = num;
            c++;
        }
    }
}
```

## 面试题 17.10. 主要元素

```
class Solution {
    public int majorityElement(int[] nums) {
        Arrays.sort(nums);
        int index = nums.length / 2;
        System.out.print(index);
        int left = index - 1, right = index + 1;
        int n = 1;
        while(left >= 0 && nums[left] == nums[index]){
            n++;
            left--;
        }
    }
}
```

```

    }
    while(right < nums.length && nums[right] == nums[index]){
        n++;
        right++;
    }
    return n > nums.length / 2 ? nums[index] : -1;
}
}

```

```

class Solution {
    public int majorityElement(int[] nums) {
        if(nums.length == 0)
            return -1;
        if(nums.length == 1)
            return nums[0];
        int flag = nums[0];
        int times = 1;
        for(int i = 1; i < nums.length; i++){
            if(times == 0){
                flag = nums[i];
                times++;
            }
            else{
                if(flag == nums[i])
                    times++;
                else
                    times--;
            }
        }
        int n = 0;
        for(int num : nums){
            if(num == flag)
                n++;
        }
        return n > nums.length / 2 ? flag : -1;
    }
}

```

## 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。 **冒泡变种**（数据量大了不行）

```

class Solution {
    public int[] exchange(int[] nums) {
        bubbleSorts(nums);
        return nums;
    }
}

```

```

private void bubbleSorts(int[] nums){
    for(int i = 0; i < nums.length - 1; i++){
        for(int j = 0; j < nums.length - 1 - i; j++){
            if(nums[j] % 2 == 0 && nums[j + 1] % 2 == 1){
                int temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
            }
        }
    }
}

```

## 双指针

```

class Solution {
    public int[] exchange(int[] nums) {
        /* 首尾双指针法
           不保证稳定性
        */
        int left = 0, right = nums.length - 1;
        while(left < right){
            while(left < right && (nums[left] & 1) == 0){
                left++;
            }
            while(left < right && (nums[right] & 0) == 0){
                right--;
            }
            int temp = nums[left];
            nums[left] = nums[right];
            nums[right] = temp;
        }
        return nums;
    }
}

```

## 快慢指针

```

class Solution {
    public int[] exchange(int[] nums) {
        /* 首位双指针法 不必担心nums[slow] 和 nums[fast] 都是奇数， 因为
           nums[slow] 的值肯定早就被nums[fast]移动了，
           slow指向的始终是下一个奇数应该存放的位置
           不保证稳定性
        */
        int slow = 0, fast = 0;
        while(fast < nums.length){
            if((nums[fast] & 1) == 1){
                if(slow != fast){ //增加效率

```

```

        int temp = nums[fast];
        nums[fast] = nums[slow];
        nums[slow] = temp;
    }
    slow++;
}
fast++;
}
return nums;
}
}

```

## 排序后有序

```

class Solution {
    public int[] exchange(int[] nums) {
        //稳定性
        int i = 0;
        int temp = 0;
        while(i < nums.length){
            if((nums[i] & 1) == 0){ //偶数
                int j = i + 1;
                while(j < nums.length && (nums[j] & 1) != 1){
                    j++;
                }
                if(j == nums.length)
                    break;
                temp = nums[j];
                for(int k = j; k > i; k--){ //整体右移一个位置
                    nums[k] = nums[k - 1];
                }
                nums[i] = temp; //奇偶交换
            }
            i++;
        }
        return nums;
    }
}

```

## 剑指 Offer 29. 顺时针打印矩阵

子烁

```

class Solution {
    public int[] spiralOrder(int[][] matrix) {
        List<Integer> resList = new ArrayList<>();
        int m = matrix.length; //行
    }
}

```

```

        if(m == 0)
            return new int[0];
        int n = matrix[0].length;    //列
        int times = (int)Math.ceil(Math.min(m, n) / 2.0);

        int c1 = 0, c2 = n - 1;
        int r1 = 0, r2 = m - 1;

        for(int i = 0; i < times; i++){
            for(int c = c1; c <= c2; c++){
                resList.add(matrix[r1][c]);
            }
            for(int r = r1 + 1; r <= r2; r++){
                resList.add(matrix[r][c2]);
            }
            if(r1 < r2 && c1 < c2){ //如果不相交的话
                for(int c = c2 - 1; c >= c1; c--){
                    resList.add(matrix[r2][c]);
                }
                for(int r = r2 - 1; r > r1; r--){
                    resList.add(matrix[r][c1]);
                }
            }
            c1++;
            c2--;
            r1++;
            r2--;
        }
        int[] res = new int[m * n];
        int i = 0;
        for(int num : resList){
            res[i] = num;
            i++;
        }
        return res;
    }
}

```

## 【动态规划】 【滑动窗口】 滑动窗口的最大值

子烁

给定一个数组 nums 和滑动窗口的大小 k，请找出所有滑动窗口里的最大值。

双端队列

```

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        List<Integer> resList = new ArrayList<>();
        Deque<Integer> deque = new LinkedList<>();
    }
}

```



```

        for(int i = 0; i < nums.length; i++){
            //如果当前队列尾部的值比当前nums[i]小, 则弹出末尾元素, 将当前序列添加到尾部
            (如果都比当前序列小, 那他自然就成为队首元素了)
            while(!deque.isEmpty() && nums[deque.getLast()] < nums[i]){
                deque.pollLast();
            }
            deque.addLast(i);

            //如果当前队列首部元素(最大值)不在滑动窗口范围内, 则弹出
            while(!deque.isEmpty() && deque.getFirst() <= i - k){
                deque.pollFirst();
            }

            if(i >= k - 1){
                resList.add(nums[deque.getFirst()]);
            }
        }
        int[] res = new int[resList.size()];
        int n = 0;
        for(int num : resList){
            res[n] = num;
            n++;
        }
        return res;
    }
}

```

## 普通滑动窗口

```

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums.length == 0)
            return new int[0];
        if(nums.length <= k)
            return new int[] {CalMax(0, nums.length - 1, nums)};
        List<Integer> resList = new ArrayList<>();

        int left = 0, right = k - 1;
        while(right < nums.length){
            resList.add(CalMax(left, right, nums));
            left++;
            right++;
        }

        int[] res = new int[resList.size()];
        int n = 0;
        for(int num : resList){
            res[n] = num;
            n++;
        }
        return res;
    }
}

```

```

private int CalMax(int left, int right, int[] nums){
    int max = nums[left];
    for(int i = left + 1; i <= right; i++){
        if(nums[i] > max)
            max = nums[i];
    }
    return max;
}
}

```

## 动态规划

```

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int right = k - 1;
        int[] dp = new int[nums.length - k + 1];

        dp[0] = nums[0];
        for(int i = 0; i < k; i++){
            dp[0] = Math.max(dp[0], nums[i]);
        }

        int p = k;
        for(int i = 1; i < dp.length; i++, p++){
            if(dp[i - 1] == nums[i - 1]){
                dp[i] = nums[i];
                for(int j = i + 1; j < p; j++){
                    dp[i] = Math.max(dp[i], nums[j]);
                }
            }
            else{
                dp[i] = Math.max(dp[i - 1], nums[p]);
            }
        }
        return dp;
    }
}

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int right = k - 1;
        int[] dp = new int[nums.length - k + 1];
        for(int i = 0; i < k; i++){
            dp[0] = Math.max(dp[0], nums[i]);
        }

        int p = k;
        for(int i = 1; i < dp.length; i++, p++){
            //如果前一个元素就是 dp[i - 1]
            if(nums[i - 1] == dp[i - 1]){
                //重新寻找最大值
            }
        }
    }
}

```

```

        dp[i] = nums[i];
        for(int j = i + 1; j <= p; j++){
            dp[i] = Math.max(dp[i], nums[j]);
        }
    }
    else{
        //如果dp[i - 1] != nums[i - 1]说明dp[i - 1]一定在当前窗口内，而且当前
        窗口只是比前一个窗口多一个nums[p]
        //前一个窗口的最大值dp[i - 1]与此窗口的最右值相比较
        dp[i] = dp[i - 1] > nums[p] ? dp[i - 1] : nums[p];
    }
}
return dp;
}
}

```

## 【动态规划】面试题 17.16. 按摩师

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。

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。

```

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

```

## 剑指 Offer 12. 矩阵中的路径

子烁

```
class Solution {
    private int[][] visited;
    public boolean exist(char[][] board, String word) {
        char[] str = word.toCharArray();
        int rows = board.length; //行数
        int cols = board[0].length; //列数
        visited = new int[rows][cols];

        for(int x = 0; x < rows; x++){
            for(int y = 0; y < cols; y++){
                if(findNext(board, rows, cols, x, y, str, 0)){
                    return true;
                }
            }
        }
        return false;
    }

    public boolean findNext(char[][] matrix, int rows, int cols, int x, int y,
        char[] str, int k){
        if(k == str.length)
            return true;

        if(x < 0 || y < 0 || x >= rows || y >= cols || visited[x][y] == 1)
            return false;
        if(str[k] != matrix[x][y])
            return false;

        visited[x][y] = 1;
        if(findNext(matrix, rows, cols, x + 1, y, str, k + 1))
            return true;
        if(findNext(matrix, rows, cols, x, y + 1, str, k + 1))
            return true;
        if(findNext(matrix, rows, cols, x - 1, y, str, k + 1))
            return true;
        if(findNext(matrix, rows, cols, x, y - 1, str, k + 1))
            return true;

        //因为在递归的过程中如果不恢复，会对其他分支产生干扰
        visited[x][y] = 0; //匹配的，即都没有访问到
        return false;
    }
}
```