贪心算法01

> 个人感觉**贪心算法其实就是没有什么规律可言**，纯纯凭感觉，不用花心思去研究其规律，**没有思路就立刻看题解**。基本贪心的题目要不就是特简单，要不就是死活想不出来，拼尽全力无法战胜。

理论基础

贪心的本质是选择每一阶段的局部最优，从而达到全局最优。

说实话贪心没有固定的套路，个人觉得一个比较好用的思想就是举反例，**如果想不到反例，就试一试贪心吧**

实战

455.分发饼干

简单 15min

题目

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 `i`，都有一个胃口值 `g[i]`，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 `j`，都有一个尺寸 `s[j]` 。如果 `s[j] ≥ g[i]`，我们可以将这个饼干 `j` 分配给孩子 `i` ，这个孩子会得到满足。你的目标是满足尽可能多的孩子，并输出这个最大数值。

示例 1:

```
输入: g = [1,2,3], s = [1,1]
输出: 1
解释:
你有三个孩子和两块小饼干，3 个孩子的胃口值分别是: 1,2,3。
虽然你有两块小饼干，由于他们的尺寸都是 1，你只能让胃口值是 1 的孩子满足。
所以你应该输出 1。
```

示例 2:

```
输入: g = [1,2], s = [1,2,3]
输出: 2
解释:
你有两个孩子和三块小饼干，2 个孩子的胃口值分别是 1,2。
你拥有的饼干数量和尺寸都足以让所有孩子满足。
所以你应该输出 2。
```

提示:

- `1 ⩽ g.length ⩽ 3 * 104`
- `0 ⩽ s.length ⩽ 3 * 104`
- `1 ⩽ g[i], s[j] ⩽ 231 - 1`

题解

一个简单的贪心+双指针，和常识一样，最小的饼干给能喂饱的胃口最小的小朋友（或者反过来，最大的给胃口最大的）

C#

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        ranges::sort(g);
        ranges::sort(s);//先给两个数组排序
        int i=0;//记录现在分到哪个小朋友手里了
        for(int x:s){//遍历每块饼干
            if(i < g.size() && g[i]<=x){//如果还有小朋友没分到饼干，且当前饼干可以满足小朋友胃口
                i++;//把这个饼干分给小朋友
            }
            //因为胃口是升序排列的，如果满足不了当前的胃口，显然也满足不了后面的，因此这里不用写循环，写个if就好，直接找下一块
        }
        return i;
    }
};
```

复杂度分析

- 时间复杂度：$O(nlogn + mlogm)$，其中 n是 g的长度，m是s的长度。瓶颈在排序上。
- 空间复杂度：$O(1)$。忽略排序的栈开销。

376. 摆动序列

中等 30min

题目

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为 **摆动序列** 。第一个差（如果存在的话）可能是正数或负数。仅有一个元素或者含两个不等元素的序列也视作摆动序列。

- 例如， `[1, 7, 4, 9, 2, 5]` 是一个 **摆动序列** ，因为差值 `(6, -3, 5, -7, 3)` 是正负交替出现的。
- 相反， `[1, 4, 7, 2, 5]` 和 `[1, 7, 4, 5, 5]` 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

子序列 可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 `nums` ，返回 `nums` 中作为 **摆动序列** 的 **最长子序列的长度** 。

示例 1:

```
输入：nums = [1,7,4,9,2,5]
输出：6
解释：整个序列均为摆动序列，各元素之间的差值为 (6, -3, 5, -7, 3) 。
```

示例 2:

```
输入：nums = [1,17,5,10,13,15,10,5,16,8]
输出：7
解释：这个序列包含几个长度为 7 摆动序列。
其中一个是 [1, 17, 10, 13, 10, 16, 8] ，各元素之间的差值为 (16, -7, 3, -3, 6, -8) 。
```

示例 3:

```
输入：nums = [1,2,3,4,5,6,7,8,9]
输出：2
```

提示:

- `1 ⩽ nums.length ⩽ 1000`
- `0 ⩽ nums[i] ⩽ 1000`

题解

这题凭理解做，做完了也没觉得是用贪心做的，结果还真是贪心，这一类方法好难分类hh，看到这道题目想着就一正一反的数着呗，只不过算差值得从第三个开始算，也就是说要对长度小于2的情况特殊讨论，之后就反一次就加一好了

01. 首先检查数组的大小，如果数组元素个数小于 2，直接返回元素个数作为摆动序列的长度。
02. 计算数组前两个元素的差值 `prevDiff`，并根据这个差值是否为 0 来初始化摆动序列的长度 `length`。如果差值为 0，长度为 1；否则为 2。
03. 然后遍历数组中从第三个元素开始的所有元素，计算当前元素和前一个元素的差值 `diff`。

04. 使用条件`(prevDiff ≤ 0 && diff > 0) || (prevDiff ≥ 0 && diff < 0)`来判断当前差值和前一个差值的正负性是否不同。如果不同，说明找到了一个新的摆动点，摆动序列的长度增加 1，并且更新`prevDiff`为当前的`diff`。

C#

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
    int n = nums.size();
    if (n < 2) {
        return n;  // 如果数组元素个数小于2，那么摆动序列的长度就是元素个数本身
    }
    int prevDiff = nums[1] - nums[0];
    int length = prevDiff == 0? 1 : 2;  // 根据前两个元素的差值是否为0来初始化摆动序列的长度

    for (int i = 2; i < n; ++i) {
        int diff = nums[i] - nums[i - 1];
        if ((prevDiff <= 0 && diff > 0) || (prevDiff >= 0 && diff < 0)) {
            // 如果当前差值和前一个差值的正负性不同
            length++;
            prevDiff = diff;  // 更新前一个差值
        }
    }
    return length;
}
};
```