回溯结束了，贪心正式开始，你准备好了嘛?

贪心算法的本质?

贪心算法的本质是选择**每一阶段**的**局部最优**，从而达到**全局最优**。

贪心算法的套路(什么时候用贪心)

说实话贪心没有套路

有同学问了如何验证可不可以用贪心算法呢?

手动模拟一下且如果想不到反例，那么就试一试贪心吧。

总结:

不好意思了，贪心没有套路，说白了就是常识性推导加上举反例

贪心算法的第一道题，就看看你够不够贪心

455.分发饼干

题目链接：https://leetcode-cn.com/problems/assign-cookies/

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。你的目标是尽可能满足越多数量的孩子，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:

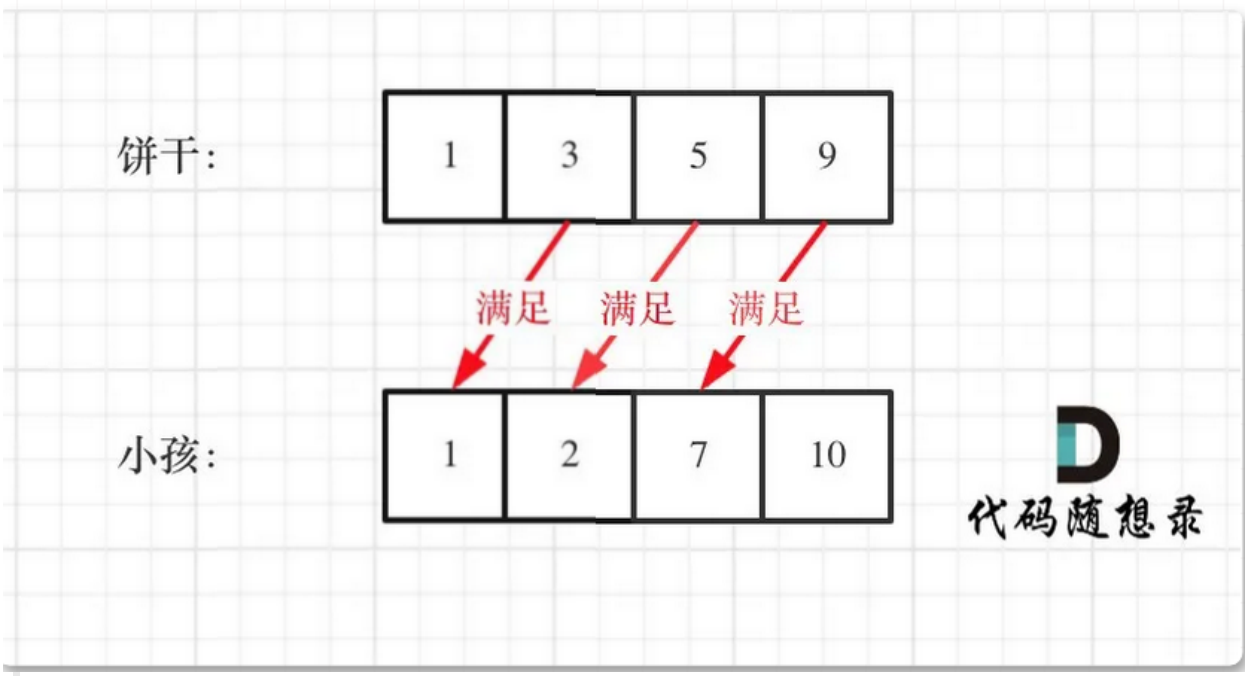
你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。

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。

> 题意中,大饼干既可以满足胃口大的孩子也可以满足胃口小的孩子，那么就应该满足胃口大的，这里的**局部最优**就是大饼干喂给胃口大的,充分利用饼干尺寸喂饱一个，**全局最优**就是喂饱尽可能多的小孩

前序就是对饼干和小孩进行排序,



思路:
对饼干数组从数组末尾向前遍历(**外层循环**)
在确定了某个饼干时,再遍历小孩数组从末尾向前遍历(**内层循环**)

```
class Solution {
public:
 //g表示小孩所需要的饼干尺寸,s表示饼干的尺寸
    int findContentChildren(vector<int>& g, vector<int>& s) {
 sort(g.begin(),g.end());//从大到小排序
 sort(s.begin(),s.end());
 int count=0,startIndex=g.size()-1;
 for(int i=s.size()-1;i>=0;i--){
 for(int j=startIndex;j>=0;j--){
  if(g[j]<=s[i]){
  count++;
  startIndex=j-1;//此时j以及满足条件了，然后就需要往前移动一位.
  break;
  }
  }
  }
  return count;
     }
};
//将二层循环转换为一层循环的写法
int index=s.size()-1;
int result=0;
for(int i=g.size()-1;i>=0;i--){
 if(index>=0&&g[i]<=s[index]){
```

```
25    index--;
26    result++;
27    }
28  }
29
```

376. 摆动序列

题目链接：https://leetcode-cn.com/problems/wiggle-subsequence/

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。少于两个元素的序列也是摆动序列。

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。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

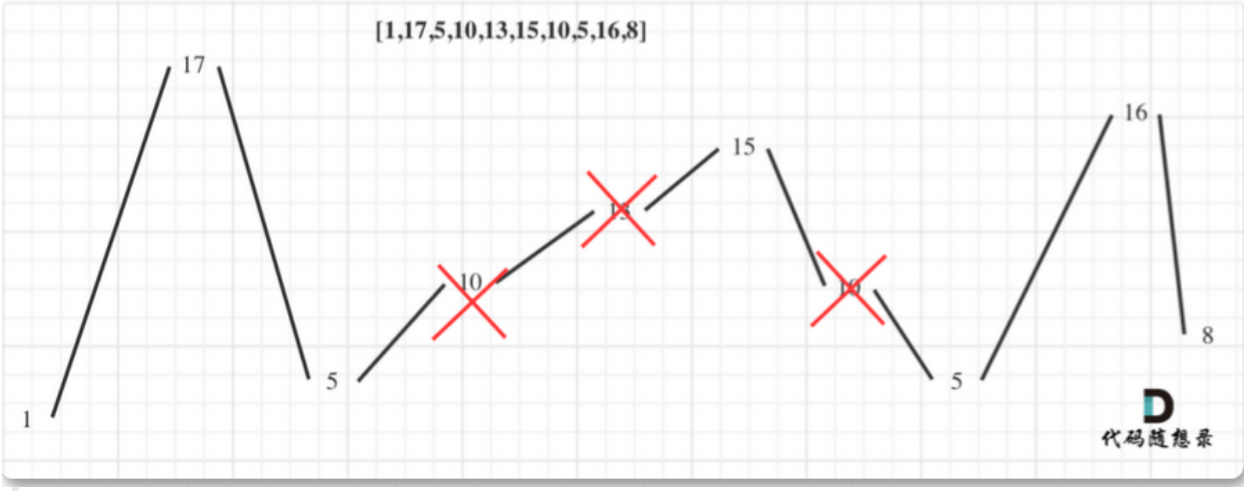
示例 1:

输入: [1,7,4,9,2,5]

输出: 6

解释: 整个序列均为摆动序列。

本题要求可以从**序列中**删除一些元素来获得**子序列,剩下的元素保持其原始顺序.**



如何选取元素呢?

一个元素到下一个元素只有递增和递减的趋势.

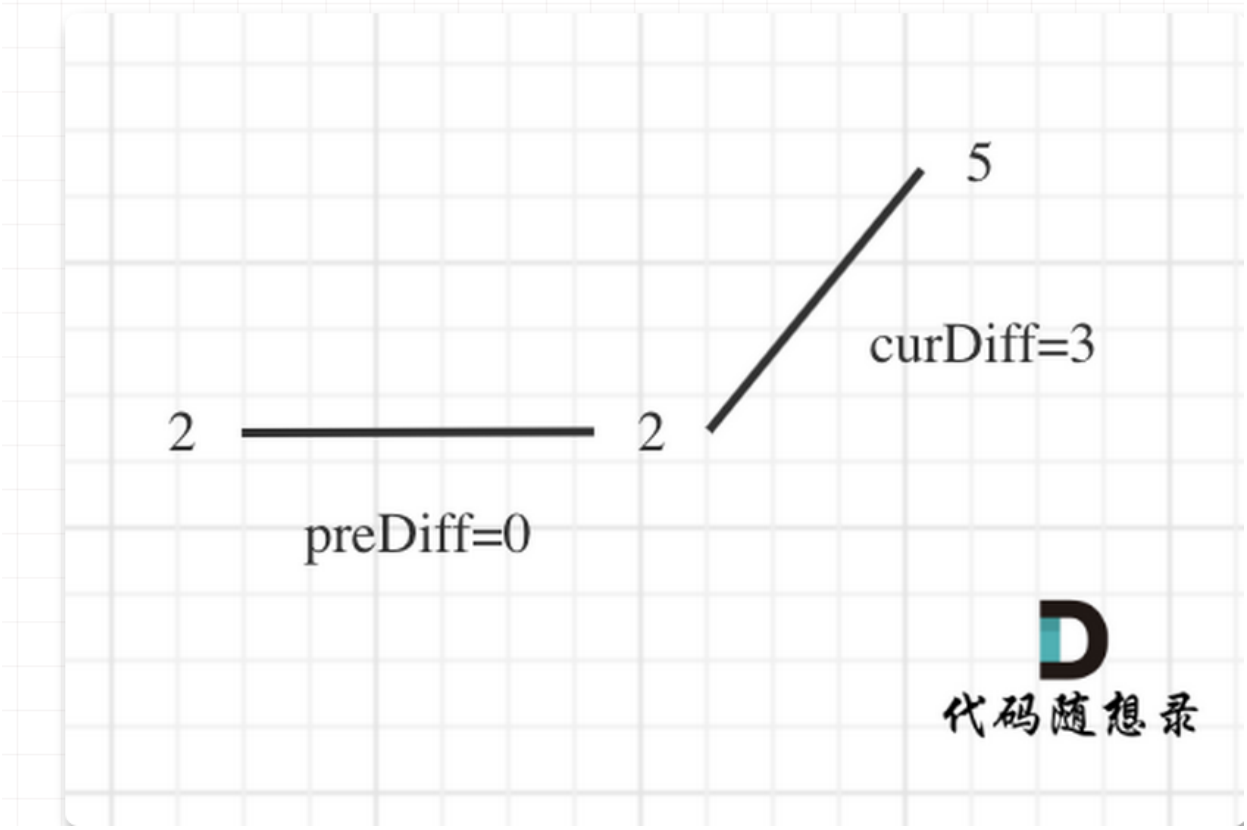
如果呈现递增的态势取最大的元素,

如果呈现递减的态势取最小的元素.

> **所以只要计算数组的峰值数量就可以了**

统计峰值的方法

> int curDiff,preDiff;//当前差值，前一个差值
> if((curDiff>0&&preDiff<=0)||(curDiff<0&&preDiff>=0)) count++;
> 在起始时候predDiff=0，假设在index=-1处有一个nums[-1]==nums[0];




```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
 int result=1;//默认最右边有一个峰值
 int curDiff=0,preDiff=0;
 for(int i=1;i<nums.size();i++){
 curDiff=nums[i]-nums[i-1];//计算当前差值
 if((curDiff<0&&preDiff>=0)||(curDiff>0&&preDiff<=0)){
//为什么这语句要放在if里面是为了防止1223这样的序列，该题答案为2如果把preDiff=curDiff放在if外面
//则答案为3，因此需要把preDiff=curDiff放在if里面
 preDiff=curDiff;
 result++;
 }
 }
 return result;
    }
};
```

53. 最大子序和

题目地址：https://leetcode-cn.com/problems/maximum-subarray/

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

输入: [-2,1,-3,4,-1,2,1,-5,4]

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

有一个种暴力算法结果为利用前缀和

然后用双指针计算[i:j]之间的和值为[0:j]-[0:i]+S[i]

本题目用动态规划更好理解

dp[i]表示以第i个元素结束的最大子序列和;

dp[i]=max(dp[i-1]+nums[i],nums[i]);

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
 if(nums.size()==0) return 0;
 vector<int> dp;
 dp.push_back(nums[0]);
 for(int i=1;i<nums.size();i++){
 int tmp=max(dp.back()+nums[i],nums[i]);
 dp.push_back(tmp);
  }
  int sum=dp.back();
  for(int i=dp.size()-2;i>=0;i--) sum=max(sum,dp[i]);
  return sum;
    }
};
```

122.买卖股票的最佳时机II

题目链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-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所能获取的最大利润。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）。

示例 1:

输入: [7,1,5,3,6,4]

输出: 7

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4。随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。

> 每次递增的部分都要计算即为最大的利润,
> **「这道题目理解利润拆分是关键点！」** 不要整块的去看，而是把整体利润拆为每天的利润，就很容易想到贪心了。
> 局部最优：只收集每天的正利润,全局最优：得到最大利润

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
 int sum=0;
 for(int i=1;i<prices.size();i++){
 if(prices[i]>prices[i-1])sum+=(prices[i]-prices[i-1]);
 }
 return sum;
    }
};
```

55. 跳跃游戏

题目链接：https://leetcode-cn.com/problems/jump-game/

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1，然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ，所以你永远不可能到达最后一个位置。

如何理解"**数组中的每个元素**代表**你在该位置可以跳跃**的**最大长度**"？

> 即你站在该位置可以跳的范围<=nums[i]
> 那么这个问题就转换为跳跃覆盖范围究竟可不可以覆盖到终点？
> 每次移动取最大跳跃步数(得到最大的覆盖范围)，每移动一个单位，就更新最大覆盖范围

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
 if(nums.size()==0) return true;
 int sumStep=nums[0];
 for(int i=1;i<nums.size()&&i<=sumStep;i++){
 sumStep=max(sumStep,i+nums[i]);
 }
 return sumStep>=(nums.size()-1)?true:false;
    }
};
```

1005.K次取反后最大化的数组和

题目地址：https://leetcode-cn.com/problems/maximize-sum-of-array-after-k-negations/

给定一个整数数组 A，我们只能用以下方法修改该数组：我们选择某个索引 i 并将 A[i] 替换为 -A[i]，然后总共重复这个过程 K 次。（我们可以多次选择同一个索引 i。）

以这种方式修改数组后，返回数组可能的最大和。

示例 1：输入：A = [4,2,3], K = 1
输出：5
解释：选择索引 (1,) ，然后 A 变为 [4,-2,3]。

示例 2：
输入：A = [3,-1,0,2], K = 3
输出：6
解释：选择索引 (1, 2, 2) ，然后 A 变为 [3,1,0,2]。

> 对K次取反我们取反的一定是**每次取数值最小的(每次取的时候都要先排序一下然后再取最小的取反)**才能**使K次取反后数组为最大化和**

```
class Solution {
public:
    int largestSumAfterKNegations(vector<int>& A, int K) {
 if(A.empty()) return 0;
 while(K>0){
 sort(A.begin(),A.end());
 A[0]=-1*A[0];
 K--;
 }
  for(int i=0;i<A.size();i++)sum+=A[i];
  return sum;
     }
};
```

134. 加油站

题目链接：https://leetcode-cn.com/problems/gas-station/

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

- 如果题目有解，该答案即为唯一答案。
- 输入数组均为非空数组，且长度相同。
- 输入数组中的元素均为非负数。

示例 1:

输入:

```
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]
```

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油

开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油

开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油

开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油

开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

示例 2:

输入:

```
gas  = [2,3,4]
cost = [3,4,3]
```

输出：-1
解释：
你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。
我们从 2 号加油站出发，可以获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油
开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油
你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。
因此，无论怎样，你都不可能绕环路行驶一周。

> 方法1就是通过模拟跑一圈的过程：
> 解的条件:如果**跑了一圈**，**中途没有断油**，而且**最后油量大于等于0**，**说明这个起点ok**
> **for循环**适合模拟从头到尾的遍历,而**while循环**适合模拟环形遍历

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
 for(int i=0;i<cost.size();i++){
 int rest=gas[i]-cost[i];//cost[i]表达从加油站i到加油站i+1所花费cost
 int index=(i+1)%cost.size();
 while(rest>=0&&index!=i){
 rest+=gas[index]-cost[index];
 index=(index+1)%cost.size();
  }
  //终止条件就是重新回到起点且汽油大于等于0
  if(rest>=0&&index==i) return 1;
  }
  return -1;
     }
};
```

贪心算法(方法二)

> 首先如果**总油量**减去**总消耗大于等于0**那么一定可以**跑完一圈**，说明各个站点的加油站剩余量rest[i]相加一定是大于等于0的。
> 每个加油站的剩余量**rest[i]=gas[i]-cost[i]**;
> i从0开始累加**rest[i]**，和记为**curSum**,一旦curSum小于0,
> 为什么“说明[0,i]区间不能作为起始位置，起始位置从i+1算起,再从0计算curSum”？.
> 因为从0号加油站到i号加油站时,curSum>0,那么可以到达i位置,当从i号加油站到i+1号加油站时,curSum<0，那么就不能到达i+1号加油站.如果从j(j<i)号加油站开始那么从0号加油站到j号的加油站时,curSum>0,此时从j号加油站到i+1号加油站时此时的sum一定小于0,所以该结论成立.

> 「局部最优:当前累加rest[j]的和curSum一旦小于0，起始位置至少要是j+1，因为从j开始一定不行。全局最优:找到可以跑一圈的起始位置

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
 int totalSum=0;
 int curSum=0;
 int start=0;
 for(int i=0;i<gas.size();i++){
 curSum+=gas[i]-cost[i];
 totalSum+=gas[i]-cost[i];
  if(curSum<0){
  curSum=0;
  start=i+1;
  }
  }
  if(totalSum<0) return -1;
  return start;
     }
};
```

135. 分发糖果

链接：https://leetcode-cn.com/problems/candy/

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻的孩子中，评分高的孩子必须获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1:

输入: [1,0,2]

输出: 5

解释: 你可以分别给这三个孩子分发 2、1、2 颗糖果。

示例 2:

输入: [1,2,2]

输出: 4

解释: 你可以分别给这三个孩子分发 1、2、1 颗糖果。

第三个孩子只得到 1 颗糖果，这已满足上述两个条件。

这道题目一定是要确定一边之后，再确定另一边,例如比较每一个孩子的左边，然后再比较右边,[如果两边一起考虑一定会顾此失彼]

1、先考虑从左到右的情况(也就是从前向后遍历)

局部最优：只要右边评分比左边大,右边的孩子就多了一个糖果,

`if(ratings[i]>ratings[i-1])candyVec[i]=candyVec[i-1]+1;`

全局最优：相邻的孩子中，评分高的右孩子获得比左边孩子更多的糖果

此时分配糖果的数组的元素值candyVec[i]与candyVec[i-1]满足题目要求

2、后考虑从右到左的情况(从后向右遍历)

局部最优,如果左边的评分比右边的大，则左边的孩子糖果数=max(原糖果数,右边孩子糖果数+1);

全局最优：满足题目条件

```
class Solution {
public:
    int candy(vector<int>& ratings) {
 vector<int> candyVec(ratings.size(),1);
 //从前向后遍历
 for(int i=1;i<ratings.size();i++){
 if(ratings[i]>ratings[i-1])
 candyVec[i]=candyVec[i-1]+1;
```

```
}
 //从后向前遍历
 for(int i=ratings.size()-2;i>=0;i--){
 if(ratings[i]>ratings[i+1])
 candyVec[i]=max(candyVec[i],candyVec[i+1]+1);
 }
 int sum=0;
 for(int i=0;i<candyVec.size();i++){
 sum+=candyVec[i];
 }
 return sum;
    }
};
```