贪心算法

贪心算法这块，很多题解的充分必要性比较难以捉摸，没有动态规划或者DFS回溯那么好想，动态规划和DFS回溯的典型特点是可以把定义域所有情况通过表或者树形结构表示出来，然后在定义域里面搜索即可。

但是贪心算法没办法表示在哪个集合里面搜索，只能按照题目要求一步一步做，每步做的时候都会进行一定的选择，这个局部选择倾向于让当前任务成功，当前任务让我跳的更远，那么我就要跳的最远，当前任务要我多给旁边孩子一颗糖，我就多给他一颗糖。

如果总任务是能不能完成一件事，通过每步最优选择做成的话，肯定能做成，如果中间哪步怎么做都不行的话，那肯定不行，这种情况实际上已经考虑了所有的方案，属于在全集中搜索了，充分必要性显然。

但是如果是求解最优问题，最少多少个糖，或者其它什么，想证明局部最优就是全局最优不是很简单的事情。

这种情况，通常如果想不通充要条件是什么，那么它的解题做法你只能先看答案，不看答案想不到怎么做，因为答案在哪个集合里面搜索你都不清楚。

所以动态规划和DFS回溯实际上是更简单的题型，只要知道在哪个集合搜索，后面的分析基本都是套路。

所以这块这里只是简单解几个题，能想通其充要性就想通，想不通算了，就当背题了。

455. 分发饼干

简单 相关标签 相关企业

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。

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.

leetcode

思路分析：

先对饼干和胃口排个序，大号饼干优先满足大胃口，这个很显然，如果大号饼干先满足一个小胃口的，那个大胃口的就没有饼干可以满足了。

```
class Solution {
public:
```

```
    int findContentChildren(vector<int>& g, vector<int>& s) {
        int result = 0;
        sort(g.begin(),g.end());
        sort(s.begin(),s.end());
        for (int i = s.size()-1, j = g.size()-1; i >= 0 && j >= 0;){
            if (s[i] >= g[j]) {
                result++;
                i--;j--;  // 满足的话,两指针一起向前移动
            }
            else j--; // 不满足,则减小胃口
        }
        return result;
    }
};
```

376. 摆动序列

中等 相关标签 相关企业 A文

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：

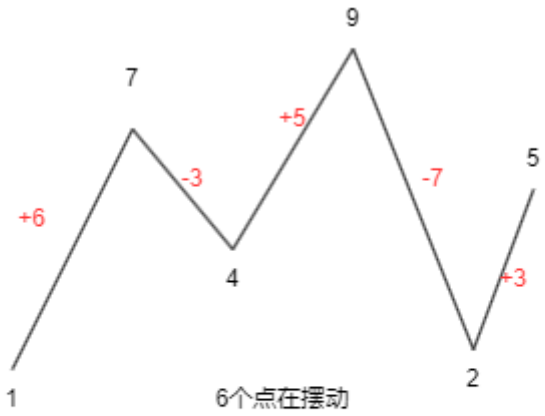
> **输入：** nums = [1,17,5,10,13,15,10,5,16,8]
> **输出：** 7
> **解释：** 这个序列包含几个长度为 7 摆动序列。
> 其中一个是 [1, 17, 10, 13, 10, 16, 8] ，各元素之间的差值为 (16, -7, 3, -3, 6, -8) 。

示例 3：

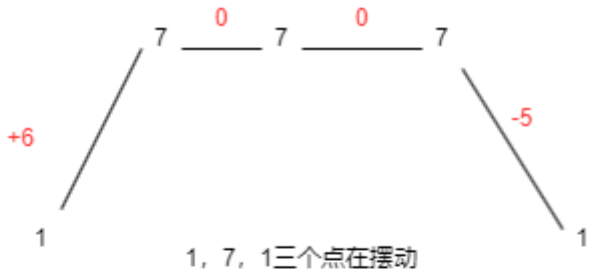
> **输入：** nums = [1,2,3,4,5,6,7,8,9]
> **输出：** 2

leetcode

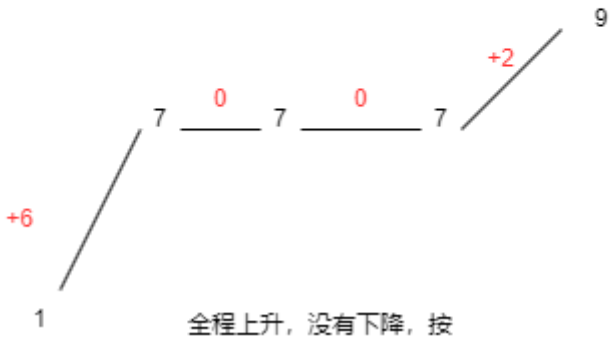
思路分析：



6个点在摆动



1，7，1三个点在摆动



全程上升，没有下降，按1个点摆动

对于示例1的序列，全部的数字构成一个摆动序列，一上一下摆动，我们只要统计拐点的数量(包括首尾)，拐点的地方两侧梯度异号，然后下面两种情况，包含平坡，比如1-7-7这里三个数字不是摆动序列，虽然有拐点，但是题目不是说出现拐点就是摆动，题目说一上一下才是摆动，所以我们需要把坡度为0的地方去掉再看。对于第三个图也是一样的道理，把0去掉（其实就是把首尾7合并了），然后再找摆动序列。

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        if (nums.size() == 0 || nums.size() == 1)
            return nums.size();
        // 求梯度
        vector<int> gradient(nums.size()-1,0);
        for (int i = 0; i < nums.size()-1; i++)
            gradient[i] = nums[i+1] - nums[i];
        // 把梯度为0的删除
        int slow = 0, fast = 0;
        for (; fast < gradient.size();){
            if (gradient[fast]!=0) gradient[slow++]=gradient[fast++];
            else fast++;
        }
        gradient.resize(slow);
```

```
        // 如果删除平坡后,彻底就没了,说明原来整个序列就是平的
        if(gradient.size()==0) return 1;

        int result = 2;
        for (int i = 0; i+1 < gradient.size(); i++){
            if ((gradient[i]*gradient[i+1]<0))
                result++;
        }
        return result;
    }
};
```

53. 最大子数组和

中等 相关标签 相关企业 A文

给你一个整数数组 `nums` ，请你找出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。

子数组是数组中的一个连续部分。

示例 1：

> **输入：** nums = [-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 。

示例 2：

> **输入：** nums = [1]
> **输出：** 1

示例 3：

> **输入：** nums = [5,4,-1,7,8]
> **输出：** 23

leetcode

这题既是贪心又是动态规划，思路一样，就是设dp[i]为以i结尾的连续子数组的最大和，然后怎么求呢，dp[i-1]可以求出来，然后就看nums[i]要不要加上前面的以i-1为结尾的连续子数组，如果加上能变大，那肯定加上，如果加上变小了，那肯定不要加。

动态规划代码

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp(n,0);
        dp[0] = nums[0];
        for (int j = 1; j < n; j++){
            dp[j] = max(dp[j-1]+nums[j],nums[j]);
        }
        int maxSum = INT_MIN;
        for (int j = 0; j < n; j++){
            maxSum = max(maxSum,dp[j]);
```

```
        }
        return maxSum;
    }
};
```

贪心代码

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp(n,0);
        dp[0] = nums[0];
        for (int j = 1; j < n; j++){
            if (dp[j-1]>0) dp[j] = dp[j-1]+nums[j];
            else dp[j] = nums[j];
        }
        int maxSum = INT_MIN;
        for (int j = 0; j < n; j++){
            maxSum = max(maxSum,dp[j]);
        }
        return maxSum;
    }
};
```

注意，上面的贪心代码不够贪心，偏动态规划，因为用了dp数组保存局部最优值，实际上可以节省空间，仅用一个变量存储最大的那个局部最优值。就是一直往后加，如果加出来是正的，更新最优值，继续往后加，如果加出来是负的，立刻舍弃，从下一个数字开始重新搜索局部最优值。实际上过程和上面动态规划背后做的是一件事情。

122. 买卖股票的最佳时机 II

中等 相关标签 相关企业

给你一个整数数组 `prices` ，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以先购买，然后在 **同一天** 出售。

返回 *你能获得的* ***最大*** *利润* 。

示例 1：

> **输入：** prices = [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 。
> 总利润为 4 + 3 = 7 。

示例 2：

> **输入：** prices = [1,2,3,4,5]
> **输出：** 4
> **解释：** 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5 - 1 = 4 。
> 总利润为 4 。

示例 3：

> **输入：** prices = [7,6,4,3,1]
> **输出：** 0
> **解释：** 在这种情况下，交易无法获得正利润，所以不参与交易可以获得最大利润，最大利润为 0 。

[leetcode]

思路分析：

先计算相邻两天买卖股票是赚钱还是亏钱，把所有赚钱的加一起就是最大盈利。

比如示例2中，

周一以1的价格买入，周二再以2的价格售出，赚1，就是price[2] - price[1]

周二买周三卖，赚1，就是price[3] - price[1]

，... ，周四买周五卖，赚1，，就是price[5] - price[4] .

总共赚4 , 就是

（price[5] - price[4]） + （price[4] - price[3]） + （price[3] - price[2]） + （price[2] - price[1]）

= price[5] - price[1] = 4

相当于周一买，周五卖。这是连续买卖的情况。

如果不是连续几天买卖，就按照上面只要能赚就买卖的原则就行了。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        // 计算并同时收集所有正利润
        int result = 0;
        for (int i = 0; i < prices.size() - 1; i++){
            int profit = prices[i+1] - prices[i];
            if (profit > 0) result+=profit;
        }
        return result;
    }
};
```

55. 跳跃游戏

中等 相关标签 相关企业

给你一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。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下标，如果可以，返回 `true` ；否则，返回 `false` 。

示例 1：

> **输入：** nums = [2,3,1,1,4]
> **输出：** true
> **解释：** 可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

示例 2：

> **输入：** nums = [3,2,1,0,4]
> **输出：** false
> **解释：** 无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。

leetcode

思路分析：

注意，这里给的数字是能跳跃的最大步长，而不是一定要跳跃那个数，站在下标i所能跳到的最远位置就是i+nums[i]，但是在i和i+nums[i]中间可能还有位置能跳的更远，因此需要遍历所有下标，记录所有下标能跳到的

最远位置，用dp数组记录每个下标的位置能跳多远，最后看有没有能跳抵达或者超过终点的即可。

不过为了节省内存，这里直接定义一个jump保存i和i之前各个位置所能跳到的最远位置。

注意，前一段区间所能跳跃的最大下标如果还在这段区间内，也就是jump<i时，说明无法抵达i，这时候无法从i起跳，因此jump落后于i的时候，不需要更新jump，绝对跳不过去了。

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

45. 跳跃游戏 II

中等 相关标签 相关企业 A文

给定一个长度为 `n` 的 **0 索引**整数数组 `nums`。初始位置为 `nums[0]`。

每个元素 `nums[i]` 表示从索引 `i` 向前跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处:

- `0 <= j <= nums[i]`
- `i + j < n`

返回到达 `nums[n - 1]` 的最小跳跃次数。生成的测试用例可以到达 `nums[n - 1]`。

示例 1:

> **输入:** nums = [2,3,1,1,4]
> **输出:** 2
> **解释:** 跳到最后一个位置的最小跳跃数是 `2`。
> 从下标为 `0` 跳到下标为 `1` 的位置，跳 `1` 步，然后跳 `3` 步到达数组的最后一个位置。

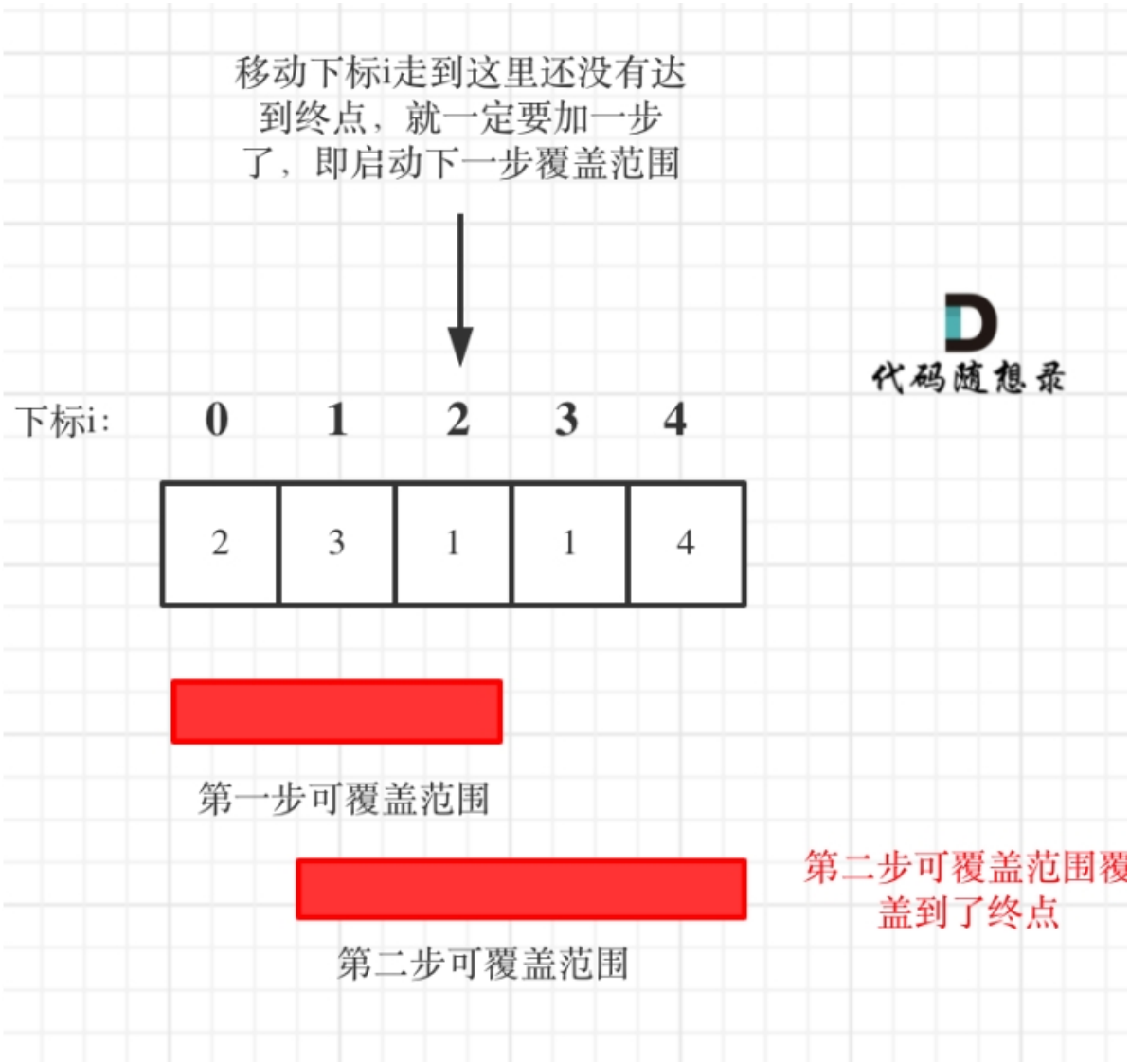
示例 2:

> **输入:** nums = [2,3,0,1,4]
> **输出:** 2

leetcode

思路分析：

主要把每次跳跃的覆盖范围表达出来，并且能够在这个覆盖范围内找到下一次从哪个位置开始跳




```
class Solution {
public:
    // 从start到end这个范围内寻找新的出发点，保证从新出发点出发能跳的更远
    int selectStart(const vector<int>& nums, int start, int end){
        int newstart = start;
        int maxJump = end;
        for (int i = start; i <= end; i++){
            if ((i+nums[i]) > maxJump){
                maxJump = i + nums[i];
                newstart = i;
            }
        }
        return newstart;
    }
    int jump(vector<int>& nums) {
        if (nums.size()==1) return 0;

        int steps = 1;
        int start = 0;
        int end = start + nums[start];

        while (true) {
            if (end >= nums.size()-1) return steps;
            start = selectStart(nums,start,end);
```

```
            end = start + nums[start];
            steps++;
        }
    }
};
```

1005. K 次取反后最大化的数组和

简单 相关标签 相关企业

给你一个整数数组 `nums` 和一个整数 `k` ，按以下方法修改该数组：

- 选择某个下标 `i` 并将 `nums[i]` 替换为 `-nums[i]` 。

重复这个过程恰好 `k` 次。可以多次选择同一个下标 `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：** nums = [4,2,3], k = 1
> **输出：** 5
> **解释：** 选择下标 1 ，nums 变为 [4,-2,3] 。

示例 2：

> **输入：** nums = [3,-1,0,2], k = 3
> **输出：** 6
> **解释：** 选择下标 (1, 2, 2) ，nums 变为 [3,1,0,2] 。

示例 3：

> **输入：** nums = [2,-3,-1,5,-4], k = 2
> **输出：** 13
> **解释：** 选择下标 (1, 4) ，nums 变为 [2,3,-1,5,4] 。

leetcode

思路分析：

先从小到大排序，画出函数图像，左边在坐标轴下面，右边在坐标轴上面，优先把最左边的值搬上来。

设负数一共有negative个，

如果k小于等于负数的值，就挨个把k个负数搬上来即可；

如果k大于负数的值，先把负数的值全部搬上来，然后看（k-negative）是奇数还是偶数，如果是偶数，

就不用对任何一个数字变号了，因为变来变去会变回来，如果是奇数，对绝对值最小的那个数变号。

```
class Solution {
public:
```

```
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        sort(nums.begin(),nums.end());
        int negative = 0;
        for (int i = 0; i < nums.size();i++){
            if (nums[i] < 0) negative++;
        }

        int result = 0;

        if (k <= negative) {
            for (int i = 0; i < k; i++) result+=-nums[i];
            for (int i = k; i < nums.size(); i++) result+=nums[i];
        }
        else {
            k = k - negative;
            for (int i = 0; i < negative; i++) nums[i]=-nums[i];
            if (k%2==0) {
                for (int i = 0; i < nums.size(); i++) result+=nums[i];
            }
            else {
                sort(nums.begin(),nums.end());
                result-=nums[0];
                for (int i = 1; i < nums.size(); i++) result+=nums[i];
            }
        }

        return result;
    }
};
```

134. 加油站

中等 相关标签 相关企业 A文

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。

给定两个整数数组 `gas` 和 `cost` ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 `-1` 。如果存在解，则 **保证** 它是 **唯一** 的。

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。

leetcode

思路分析：

首先把从每个加油站出发，油量变化量表示出来，只能从油量变化量为正数（包含0）的加油站出发，不然抵达下一个加油站，油量为负数，说明半路就没油了。

然后这题确定出发点的方式有点怪，就是先假设从第一个油量变化量为正的地方出发，然后一直开，路过的加油站需要把油量变化量相加，这就是开的过程中油箱中剩余的油量，注意油量变化量的累加值才是剩余油量，油量变化量本身不是油箱中的剩余量。直到累加值为负数的时候，停止，并以后面第一个油量变化量为正的地方出发。

然后这题是怎么想到出发点是这样的呢？

我没有思考怎么想到的，没有时间停在这题上面思考，但是可以告诉你，这个方式确定的出发点没问题，

这里说下一个才是出发点，同时就排除了前面任何一个加油站都不是出发点，认为就是从前面任何一个加油站出发，到这里剩余油量都是负数，那么你认为可能不是这样，前面可能有一家加油站可能是出发点，从那家出发到这里累加值不是负数，经过这样假设，那家之前的路程累加值一定是负数，因为全程累加值是负数。

然后你这样假设确定的那家，不还是满足，那家之前的累加值为负数吗，不又满足前面讲的新出发点的确定方式了吗。

所以这题必须先知道答案，然后才能验证这个答案没问题，至于如何循序渐进推出来的，并不明确。


```
class Solution {
```

```
public:
    int getNewStart(int start, const  vector<int> & deltaGas){
        int n = deltaGas.size();
        int newStart = start;
        int sum = 0;
        for (int i = start; i < start + n; i++){
            sum += deltaGas[i%n];
            if (sum < 0) {          // 仅碰到不满足时才会更新出发点
                newStart = i + 1;
                break;
            }
        }
        return newStart; // 可能满足要求,还是老出发点,也可能更新为新出发点了
    }
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        vector<int> deltaGas(n);
        for (int i = 0; i < n; i++)
            deltaGas[i] = gas[i] - cost[i];

        // 从第一个加油站出发,不断获取新出发点,并从新出发点再次出发
        // 新出发点前面的加油站不可能是出发点了
        int oldStart = 0;
        int newStart = 0;
        while (true) {
            newStart = getNewStart(oldStart,deltaGas);
            // 如果返回的新出发点超过n了,说明失败
            if (newStart >= n) return -1;
            // 如果返回的新出发点没变,说明成功
            if (oldStart == newStart) return newStart;
            // 不是上面两种情况,就从新出发点开始出发
            oldStart = newStart;
        }

    }
};
```

135. 分发糖果

困难 相关标签 相关企业 A文

`n` 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。

你需要按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 `1` 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。

请你给每个孩子分发糖果，计算并返回需要准备的 **最少糖果数目** 。

示例 1：

> **输入：** ratings = [1,0,2]
> **输出：** 5
> **解释：** 你可以分别给第一个、第二个、第三个孩子分发 2、1、2 颗糖果。

示例 2：

> **输入：** ratings = [1,2,2]
> **输出：** 4
> **解释：** 你可以分别给第一个、第二个、第三个孩子分发 1、2、1 颗糖果。
> 第三个孩子只得到 1 颗糖果，这满足题面中的两个条件。

[leetcode]

思路分析：

这题充要性有点难想，不像前面有的凭感觉就知道是满足充要性的，这题答案就是初始给每个孩子一个糖果，然后第一次先从左边到右边扫一遍，扫描过程中，遇到分数更高的孩子，让其糖果数量为左边孩子糖果数加1；

接着从右边往左边扫一遍，遇到分数更高的孩子，让其糖果数量为右边孩子糖果数量加1，并且同时要兼容第一次扫描这个孩子的糖果数量。

这样每个孩子的糖果数量就都不会与他旁边两位的糖果数量发生矛盾了。

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int n = ratings.size();
        vector<int> result(n,1);
        for (int i = 0; i < n - 1; i++){
            if (ratings[i+1] > ratings[i])
                result[i+1] = result[i] + 1;
        }
        for (int i = n - 1; i > 0; i--){
            if (ratings[i-1] > ratings[i]){
                result[i-1] = max(result[i-1],result[i]+1);
```

```
            }
        }
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += result[i];
        return sum;
    }
};
```

860. 柠檬水找零

简单 相关标签 相关企业

在柠檬水摊上，每一杯柠檬水的售价为 `5` 美元。顾客排队购买你的产品，（按账单 `bills` 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 `5` 美元、`10` 美元或 `20` 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 `5` 美元。

注意，一开始你手头没有任何零钱。

给你一个整数数组 `bills` ，其中 `bills[i]` 是第 `i` 位顾客付的账。如果你能给每位顾客正确找零，返回 `true` ，否则返回 `false` 。

示例 1：

> **输入：** bills = [5,5,5,10,20]
> **输出：** true
> **解释：**
> 前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
> 第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
> 第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
> 由于所有客户都得到了正确的找零，所以我们输出 true。

示例 2：

> **输入：** bills = [5,5,10,10,20]
> **输出：** false
> **解释：**
> 前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
> 对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
> 对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
> 由于不是每位顾客都得到了正确的找零，所以答案是 false。

[leetcode]

思路分析：

很简单，分3种情况：顾客给5元，顾客给10元，顾客给20元。

如果给5元，则记录5元的票子多一张；

如果给10元，看手上有没有5元，有的话找5元，并获得一张10元，没有的话失败返回；

如果给20，有两种找零方式，10+5，或者5+5+5，优先10+5的找零方式，如果这两种都无法找零，失败返回；

如果遍历一遍顾客给的票子，都没有失败返回，说明全部找零成功。

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int five = 0; int ten = 0;
        for (int i = 0; i < bills.size(); i++){
            if (bills[i] == 5) five++;
            if (bills[i] == 10) {
                if (five) {five--;ten++;}
                else return false;
```

```
            }
            if (bills[i] == 20){
                if (ten&&five) {ten--;five--;}
                else if (five>=3) {five-=3;}
                else return false;
            }
        }
        return true;
    }
};
```

763. 划分字母区间

中等 相关标签 相关企业 提示

给你一个字符串 `s` 。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。

注意，划分结果需要满足：将所有划分结果按顺序连接，得到的字符串仍然是 `s` 。

返回一个表示每个字符串片段的长度的列表。

示例 1：

> **输入：** s = "ababcbacadefegdehijhklij"
> **输出：** [9,7,8]
> **解释：**
> 划分结果为 "ababcbaca"、"defegde"、"hijhklij" 。
> 每个字母最多出现在一个片段中。
> 像 "ababcbacadefegde", "hijhklij" 这样的划分是错误的，因为划分的片段数较少。

示例 2：

> **输入：** s = "eccbbbbdec"
> **输出：** [10]

思路分析：

这题很像跳跃游戏，需要找到一个区间，让区间内任何一个字母都跳不出去，每个字母能跳到的位置就是它们最后一次出现的位置。

我们先用一个map记录所有字母最后一次出现的位置，由于只有26个字母，所以可以用vector代替map。

然后用i指针依次从头开始遍历所有位置，并拿j指针指向当前字母能跳到的最远位置，当i追到j的时候，就是分割点。记录下每个分割点就行了，然后变成题目要的长度。

```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        vector<int> jump(26);
        for (int i = 0; i < s.size(); i++)
            jump[s[i]-'a'] = i;
```

```
        vector<int> result;
        int j = 0;
        for (int i = 0; i < s.size(); i++){
            j = max(j,jump[s[i]-'a']);
            if (i == j) result.push_back(i);
        }
        for (int i = result.size()-1; i > 0; i--)
            result[i] = result[i] - result[i-1];
        result[0]++;
        return result;
    }
};
```