1. Say you have an array for which the ith element is the price of a given stock on day i.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

思想：

最大收益就是找到单调增区间，只要第 i 天比第 i-1 天的的股价高，price[i]-price[i-1]就计算在收益值中。最终的收益就是所有单调增区间的收益和。

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

2. There are *N* gas stations along a circular route, where the amount of gas at station *i* is `gas[i]`.

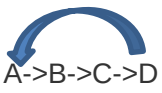
You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:

The solution is guaranteed to be unique.

用到的两个数学结论：



(1) 如果 A 可以到达 B，而 A 不可以到达 C，那么 B 也不可能到达 C。

证明：

A 能够到达 B，说明：gas[A]-cost[A]>=0

A 不能到达 C 说明：gas[A]-cost[A]+gas[B]-cost[B]<0,又因为 gas[A]-cost[A]>=0，所以 gas[B]-cost[B]<0，因此 B 不能到达 C。

(2)在整条路径中，如果

$$\sum_{i=0}^{P} gas[i] - \cos t[i] > 0$$

那么一定存在一个加油站可以从这个站出发又回到这个加油站。
所以程序实验的流程：

1、 如果 $\sum_{i=0}^{P} gas[i] - \cos t[i] < 0$，返回 -1；

2、 遍历每个站点，一旦到站点 s 时，$\sum_{i=0}^{s} gas[i] - \cos t[i] < 0$，那么从起点到站点 s 之间每个点都没办法完成，那么从第 s+1 点开始尝试。

```
class Solution {
public:
    int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
        if (gas.size() == 0 || cost.size() == 0 || gas.size() != cost.size()) return -1;
        int total = 0, sum = 0, start = 0;
        for (int i = 0; i < gas.size(); i++){
            total += (gas[i] - cost[i]);
            if (sum < 0){ //发现油箱空了，从下一个站点尝试
                sum = (gas[i] - cost[i]);
                start = i;
            }
            else{
                sum += (gas[i] - cost[i]);
            }
        }
        return total < 0 ? -1 : start; //用total判断start 是否是满足要求的解
    }
};
```

3. Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given `"bcabc"`

Return `"abc"`

Given `"cbacdcbc"`

Return `"acdb"`

解：

当一个字母出现 n 次的时候需要考虑删除哪 n-1 个，发现结果的产生和两个因素有关：第一个是字母出现的位置，第二个就是字母剩下的个数。

思路为从前先后扫描字符串，当第 i 个字母小于第 i-1 的字母序时，看第 i 个字母在后面还有没有 i-1，如果还有，就删除这个。

```
class Solution {
public:
    string removeDuplicateLetters(string s) {
        string re="";
        vector<int> counts(s.length(),0);
        vector<bool> isIn(s.length(),false);
        for (int i = 0; i<s.length(); i++) {
            counts[s[i]-'a']++;
        }
        if (s.length()==0 || s.length()==1) {
            return s;
        }
        re.push_back(s[0]);
        isIn[s[0]-'a']=true;
        counts[s[0]-'a']--;
        for (int i = 1; i<s.length(); i++) {
            if (isIn[s[i]-'a']) {
                counts[s[i]-'a']--;
                continue;
            }
            while (!re.empty() && re.back()>s[i] && counts[re.back()-'a']>0) {
                isIn[re.back()-'a'] = false;
                re.pop_back();
            }
            re.push_back(s[i]);
            isIn[s[i]-'a'] = true;
            counts[s[i]-'a']--;
        }
        return re;
    }
};
```