最佳买卖股票总结

买卖股票问题作为 LeetCode 中一类题，其思想很有指导意义，现总结如下：

1. LeetCode 121. Best Time to Buy and Sell Stock

Category	Difficulty	Likes	Dislikes
algorithms	Easy (47.34%)	2700	129

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit =
6-1 = 5.
             Not 7-1 = 6, as selling price needs to be larger than buying
price.
```

Example 2:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

问题总结下来就是一句话：一次买卖股票所能取到的最大收益。

代码：

繁琐版(自己)：

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() <= 1) {
            return 0;
        }
        int length = prices.size();
```

```
        for (int i = 0; i < length - 1; ++i) {
            prices[i] = prices[i + 1] - prices[i];
        }
        return windowmax(prices, length - 1);
    }


    int windowmax(vector<int>& nums, int length) {
        int base = nums[0];
        // int length = nums.size();
        int res = base;
        for (int i = 1; i < length; ++i) {
            if (base <= 0) {
                base = nums[i];
            }
            else {
                base += nums[i];
            }
            res = max(res, base);

        }
        return res > 0? res : 0;
    }
};
```

作为最开始想法，其时间复杂度、空间复杂度均为 `O(n)`，解决思路是：

1. 首先求取所有相邻两项之差；
2. 相邻两项只差的连续最大值即为购买一次所能取得的最大收益（毕竟中间的差值和代表以该区间首尾位置作为买卖的最大收益）。

这种思路相对比较繁琐，但是可以解决问题。

简约版：

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

        }
```

```
        return maxProfit;

    }
```

这种思路很简约，也很高效；其解题思路如下：

1. 求取遍历位置的最小值，作为买的价钱；
2. 求取遍历位置与当前记录最小值的差，作为卖的价钱；
3. 由于两者都是同步遍历，因此，这种结果可以代表遍历处所能取到的最大收益，当遍历完所有数据，即得到全局购买一次的最大收益。

2. LeetCode 122. Best Time to Buy and Sell Stock II

Category	Difficulty	Likes	Dislikes
algorithms	Easy (52.04%)	1043	1311

Say you have an array for which the *i*th 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 (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit =
5-1 = 4.
             Then buy on day 4 (price = 3) and sell on day 5 (price = 6),
profit = 6-3 = 3.
```

Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit =
5-1 = 4.
             Note that you cannot buy on day 1, buy on day 2 and sell them
later, as you are
             engaging multiple transactions at the same time. You must sell
before buying again.
```

Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

问题总结就是一句话：多次买卖股票所能取到的最大收益；

代码：

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {

        if (prices.size() <= 1) {
            return 0;
        }
        int length = prices.size();
        for (int i = 0; i < length - 1; ++i) {
            prices[i] = prices[i + 1] - prices[i];
        }
        return windowmax(prices, length - 1);

    }


    int windowmax(vector<int>& nums, int length) {
        int base = nums[0];
        int res = 0;
        for (int i = 0; i < length; ++i) {
            if (nums[i] <= 0) continue;
            res += nums[i];
        }
        return res;
    }
};
```

这道题可以直接采用第一道题的思路，因为是可以购买任意多次，所以我们只需要将所有可能的收益加上即可；

其实现的 `windowmax` 与第一题有所区别，第一题的收益是一次买卖收益，第二题的收益是任意次购买收益。

因此，同样得到所有相邻差值之后，第一题的 `windowmax` 会考虑首尾区间的收益为负的值，保证得到的收益是首尾区间的一次性交易差值；而第二题的 `windowmax` 则不会收益为负的值，而是将所有收益为正的值进行累计求和。

如对输入为 `[7,1,5,3,6,4]`，第一题的 `windowmax` 代码为：

```
    int windowmax(vector<int>& nums, int length) {
        int base = nums[0];
```

```
        // int length = nums.size();
        int res = base;
        for (int i = 1; i < length; ++i) {
            if (base <= 0) {
                base = nums[i];
            }
            else {
                base += nums[i];
            }
            res = max(res, base);

        }
        return res > 0? res : 0;
    }
```

从其代码逻辑可以看出，就是求解局部连续最大值，这时候只要 `base > 0` 就会继续进行求和，其输出结果为 5，表示以 1 购买，以 6 售出；

同样输入 `[7,1,5,3,6,4]`，对第二题来讲，第二题的 `windowmax` 代码如下：

```
    int windowmax(vector<int>& nums, int length) {
        int base = nums[0];
        int res = 0;
        for (int i = 0; i < length; ++i) {
            if (nums[i] <= 0) continue;
            res += nums[i];
        }
        return res;
    }
```

其计算的是所有的正收益累计和，直接将负收益的跳过去了(对应其可以交易多次，每个正收益都对应一次交易)，其输出结果为 7，表示以 1 买入，以 5 售出，以 3 买入，以 6 售出，最终收益为 (5-1)+(6-3) = 7。

3. LeetCode 123. Best Time to Buy and Sell Stock III

Category	Difficulty	Likes	Dislikes
algorithms	Hard (33.77%)	1112	56

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most *two*transactions.

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-
0 = 3.
             Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit
= 4-1 = 3.
```

Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-
1 = 4.
             Note that you cannot buy on day 1, buy on day 2 and sell them
later, as you are
             engaging multiple transactions at the same time. You must sell
before buying again.
```

Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

总结问题就是一句话：最多交易两次股票情况下的最大收益。

代码：

繁琐版(错误代码)：

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() < 2) return 0;
        int length = prices.size() - 1;
        for (int i = 0; i < length; ++i) {
            prices[i] = prices[i + 1] - prices[i];
        }
        return windowmax(prices, length);
    }

   int windowmax(vector<int>& prices, int length) {
        vector<int> res;
        int temp = prices[0];
        int temp_res = prices[0];
        for (int i = 1; i < length; ++i) {
            if (temp <= 0) {
                temp = prices[i];
```

```
                if (temp_res > 0) {
                    res.push_back(temp_res);
                    temp_res = 0;
                }
            }
            else {
                temp += prices[i];
            }
            temp_res = max(temp_res, temp);
        }
        res.push_back(temp_res);
        int res_len = res.size();
        if (res_len <= 1) return res[0] > 0? res[0] : 0;
        sort(res.begin(), res.end());
        return res[res_len - 1] + res[res_len - 2];

    }
};
```

这道题最初的想法依旧是采用局部差寻找两次收益的最大值，其思路如上，但是需要求两个局部最大值的最大和，其对应的 `windowmax` 函数迭代版本写了很多版，但是依旧存在问题，这个问题暂时先放下，以后有时间再填坑。

下面来分析下常规解题思路：

简约版

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() < 2) return 0;
        int length = prices.size() - 1;
        vector<int> front_profit(length + 1);
        vector<int> back_profit(length + 1);
        int valley = prices[0];
        int peak = prices[length];
        for (int i = 1; i <= length; i++) {
            front_profit[i] = max(front_profit[i-1], prices[i] - valley);
            valley = min(valley, prices[i]);
        }
        for (int j = length - 1; j >= 0; j--) {
            back_profit[j] = max(back_profit[j+1], peak - prices[j]);
            peak = max(peak, prices[j]);
        }
        int max_profit = 0;
        for (int i = 0; i < length; i++) {
            max_profit = max(max_profit, front_profit[i] + back_profit[i]);

        }
```

```
        return max_profit;
    }
};
```

前后分成两段，其本质是问题一中解法的进化版本，只是采用了迭代两次，其解题思路如下：

1. 首先得到从前向后与从后向前进行遍历得到对应位置的最佳利润；
2. 再综合考虑上述两个列表，重新进行一次遍历，得到中间各个位置作为分界点其对应前后两个遍历结果的最大利润值。

其思路清晰，代码简约，很值得学习借鉴。