股票问题大合集😂

版本1(简单

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。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

注意你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

```cpp
// 这个思路好像不太像动态规划
// 1 2 4 3 6
// 1买进6卖出==1买进2卖出 然后2又买进4卖出 然后4买进3卖出 然后3买进6卖出
// 1->4积累了3元。 4->3损失了1元 还有2元>0
// 如果 序列是 1 2 4 0 6
// 那么 4->0损失了4元 1->4积累了3元 还他妈亏了1元
// 那干脆前面的交易都算了(1->2->4)   从0开始重新积累
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int cur=0;
        int ans=0;
        for(int i=1;i<prices.size();i++){
            cur=max(0,cur+prices[i]-prices[i-1]);
            ans=max(ans,cur);
        }
        return ans;
    }
};
```

```cpp
// 这个是动态规划转换过来的
// 先看最终版本
// 然后看一步一步怎么推过来的
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(!prices.size())return 0;
        int buy=-prices[0];
        int ans=0;
        int sell=0;
        for(int i=1;i<prices.size();i++){
            sell=buy+prices[i];
            ans=max(ans,sell);
            buy=max(0-prices[i],buy);
        }
        return ans;
    }
};
/*
第一步
dp[i][1][0]表示一种状态的最大利润 ，什么状态呢?
就是当前是第i天 还有一次购买股票的机会 目前手上有0张已经购买的股票 这种状态
按照题目的意思 只有一次买卖机会 也就是购买股票的机会最大是1 手头的股票数量最大也是1
😂
*/
/*
根据上边的描述可以写出下边的表达式
首先还有一次购买机会的时候 d[i][1][0]肯定是0
d[i][0][1]表示第i天手上有一个股票，那这个状态怎么转移呢?
第i天手上的股票可能是第i天买的 也可能不是第i天买的
如果是第i天买的 dp[i][0][1]=dp[i][1][0]-prices[i]
否则说明第i-1天的时候手上就已经有一个股票了 即dp[i-1][0][1]

那么对于dp[i][0][0]呢 第i天的时候没有股票，也不能买股票
那可能第i天把股票卖了也可能是第i-1天之前的某一天股票就没有了
如果是后者 dp[i][0][0]肯定等于0
*/
for(int i=1;i<prices.size();i++){
    dp[i][1][0] = 0;// 铁定的  对于所以i dp[i][1][0]==0
    dp[i][0][1] =max( dp[i][1][0] - prices[i], dp[i-1][0][1]);
    dp[i][0][0] = dp[i-1][0][1] + prices[i];
}
/*
对上边的进行整理 既然dp[i][1][0]始终=0
就代入
*/
for(int i=1;i<prices.size();i++){
    dp[i][0][1] =max( 0 - prices[i], dp[i-1][0][1]);
    dp[i][0][0] = dp[i-1][0][1] + prices[i];
}
/*
对上边的继续整理，发现dp[i][j][k]中 j 始终是0
```

```
所以说明这个j对状态没有影响
去掉
*/
for(int i=1;i<prices.size();i++){
    dp[i][1] =max( 0 - prices[i], dp[i-1][1]);
    dp[i][0] = dp[i-1][1] + prices[i];
}
/*
继续化简
就是交换了语句的顺序，其实无所谓。
*/
for(int i=1;i<prices.size();i++){
    dp[i][0] = dp[i-1][1] + prices[i];
    dp[i][1] =max( 0 - prices[i], dp[i-1][1]);
}
/*
发现 第i天只依赖第i-1的状态
显然可以空间优化
直接用新的值覆盖老的值
*/
for(int i=1;i<prices.size();i++){
    sell=buy+prices[i];
    buy=max(0-prices[i],buy);
}
```

版本2(简单

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 。
```

示例 2:

```
输入: [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 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

状态转移方程dp[i][0]=max(dp[i-1][1]+prices[i],dp[i-1][0]) dp[i][1]=max(dp[i-1][0]-prices[i],dp[i-1][1]);

dp[i][0]表示第i天手上没有股票

dp[i][1]表示第i天手上有股票

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

发现第i天的状态和只第i-1天有关所以可以优化空间

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(!prices.size())return 0;
        int ans=0;
        int has=-prices[0];
```

```
        int none=0;
        for(int i=1;i<prices.size();i++){
            int pre_none=none;
            none=max(has+prices[i],none);
            has=max(pre_none-prices[i],has);
            ans=max(ans,none);
        }
        return ans;
    }
};
```

版本3(困难

给定一个数组，它的第 *i* 个元素是一支给定的股票在第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 *两笔* 交易。

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: [3,3,5,0,0,3,1,4]
输出: 6
解释: 在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
     随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3 。
```

示例 2:

```
输入: [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 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这个情况下, 没有交易完成, 所以最大利润为 0。
```

思路:

dp[i][j][0]==第i天还有j次购买机会手中有0个股

dp[i][j][0]=max(dp[i-1][j][1]+prices[i],dp[i-1][j][0])

dp[i][j][1]=max(dp[i-1][j+1][0]-prices[i],dp[i-1][j][1]);

化简

dp[j][0]=max(dp[j][1]+prices[i],dp[j][0]);

dp[j][1]=max(dp[j+1][0]-prices[i],dp[j][1]);

```
class Solution{
    public:
```

```
    int maxProfit(vector<int>&prices){
        if(!prices.size())return 0;
        // 恒等式dp[i][2]=0;
        vector<vector<int>>dp(3,vector<int>(2,0));
        int ans=0;
        dp[1][1]=-prices[0];
        dp[0][1]=-prices[0]; // 这个条件一定要带上 妈的坑了我好久QAQ 😂
        for(int i=1;i<prices.size();i++){
            for(int j=1;j>=0;j--){
                dp[j][0]=max(dp[j][1]+prices[i],dp[j][0]);
                dp[j][1]=max(dp[j+1][0]-prices[i],dp[j][1]);
                ans=max(ans,dp[j][0]);
            }
        }
        return ans;
    }
};
```

执行结果： 通过 显示详情 >

执行用时： 4 ms ，在所有 C++ 提交中击败了 99.57% 的用户

内存消耗： 9.7 MB ，在所有 C++ 提交中击败了 40.24% 的用户

炫耀一下：

进行下一个挑战：

买卖股票的最佳时机 IV

三个无重叠子数组的最大和

版本4(困难

给定一个数组，它的第 *i* 个元素是一支给定的股票在第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **k** 笔交易。

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: [2,4,1], k = 2
输出: 2
解释: 在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。
```

示例 2:

```
输入: [3,2,6,5,0,3], k = 2
输出: 7
解释: 在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 = 3-0 = 3 。
```

思路:k次和2次的想法是一样的 ， 替换一下变量就行了

```
class Solution{
    public:
        int maxProfit(int k, vector<int>& prices) {
            if(!prices.size())return 0;
            // 恒等式dp[i][2]=0;
            vector<vector<int>>dp(k+1,vector<int>(2,0));
            int ans=0;
            for(int i=0;i<k;i++)dp[i][1]=-prices[0];
            for(int i=1;i<prices.size();i++){
                for(int j=k-1;j>=0;j--){
                    dp[j][0]=max(dp[j][1]+prices[i],dp[j][0]);
                    dp[j][1]=max(dp[j+1][0]-prices[i],dp[j][1]);
                    ans=max(ans,dp[j][0]);
                }
            }
            return ans;
        }
};
```

但是会出现问题 通过了209个case 还有几个case通不过😂

是空间开太大了

买卖股票的最佳时机 IV

提交记录

209 / 211 个通过测试用例

状态：执行出错

提交时间：0 分钟之前

执行出错信息：	==28==AddressSanitizer's allocator is terminating the process instead of returning 0
最后执行的输入：	1000000000 [106,373,495,46,359,919,906,440,783,583,784,73,238,701,972,308,165,774,990,675,737,990,713,157,211,880,…

空间不够了，这个非常大，然后想怎么优化空间

```
for(int i=1;i<prices.size();i++){
```

```
    for(int j=k-1;j>=0;j--){
      // 快看下面这俩条诶
      // 好像dp[j][x]只和dp[j+1][x]有关诶
      // 但是不能覆盖噢，因为外边还有一层循环呢
      // 然而我傻到在考虑怎么覆盖
      // 甚至写出了代码 还能通过100多个测试点。。。。
      // 但是后来调bug的时候醒悟了
      // 但是说不定真的有办法只是我不知道QAQ
        dp[j][0]=max(dp[j][1]+prices[i],dp[j][0]);
        dp[j][1]=max(dp[j+1][0]-prices[i],dp[j][1]);
        ans=max(ans,dp[j][0]);
    }
}
```

然后看了别人的写法

发现了一个技巧

如果总共就n天 那么最多也就买卖n/2次

如果k>n/2那就可以看作是不限交易次数

于是问题转化了 转化成版本2😂

```
// 太菜了 折腾了3小时 终于好了
class Solution{
    public:
        int maxProfit(int k, vector<int>& prices) {
            if(!prices.size()||!k)return 0;
            if(k>prices.size()/2){
                cout<<"shit";
                return maxProfitUtil(prices);
            }
            // 恒等式dp[i][2]=0;
            vector<vector<int>>dp(k+1,vector<int>(2,0));
            int ans=0;
            for(int i=0;i<k;i++)dp[i][1]=-prices[0];
            for(int i=1;i<prices.size();i++){
                for(int j=k-1;j>=0;j--){
                    dp[j][0]=max(dp[j][1]+prices[i],dp[j][0]);
                    dp[j][1]=max(dp[j+1][0]-prices[i],dp[j][1]);
                    ans=max(ans,dp[j][0]);
                }
            }
            return ans;
        }
        int maxProfitUtil(vector<int>& prices) {
            if(!prices.size())return 0;
            int ans=0;
            vector<vector<int>>dp(prices.size(),vector<int>(2,0));
            dp[0][0]=0;
            dp[0][1]=-prices[0];
```

```
            for(int i=1;i<prices.size();i++){
                dp[i][0]=max(dp[i-1][1]+prices[i],dp[i-1][0]);
                dp[i][1]=max(dp[i-1][0]-prices[i],dp[i-1][1]);
                ans=max(ans,dp[i][0]);
            }
            return ans;
        }
};
```

版本5(中等

给定一个整数数组，其中第 *i* 个元素代表了第 *i* 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 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）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

示例:

```
输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

仍旧使用上面的思路

先把状态表示出来

dp[i][0]表示第i天手中没有股票 时的最大利润

dp[i][2]表示第i天为冷冻期 时的最大利润

dp[i][1]表示第i天手中有股票 时的最大利润

如果第i天手上有股票 要么就是第i-1天手上就有了，要么就是第i天刚买的 而且第i-1天不是冷冻期

于是列出dp[i][1]=max(dp[i-1][1],dp[i-1][0]-prices[i]);

如果第i 天手上没有股票 那可能是第i-1天的时候就没有，也可能是第i-2天卖了然后第i-1天是冷冻期😂（虽然冷冻期手上也没有 但理解那个意思就行

于是列出dp[i][0]=max(dp[i-1][0],dp[i-1][2]);

如果第i天是冷冻期，必然是第i-1天持有股票然后卖了

dp[i][2]=dp[i-1][1]+prices[i];

思维已经定了…… 写的时候觉得没啥问题

但是现在想想😂

为啥不是dp[i][2]=dp[i-2][1]+prices[i-1]

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(!prices.size())return 0;
        vector<vector<int>>dp(prices.size()+1,vector<int>(3,0));
        dp[0][1]=-prices[0];
        int ans=0;
        for(int i=1;i<prices.size();i++){
            dp[i][1]=max(dp[i-1][1],dp[i-1][0]-prices[i]);
            dp[i][0]=max(dp[i-1][0],dp[i-1][2]);
            dp[i][2]=dp[i-1][1]+prices[i];
            ans=max(ans,dp[i][2]);
            ans=max(ans,dp[i][0]);
        }
        return ans;
    }
};
```

看到dp[i][x]只和dp[i-1][x]有关 所以可以优化一下空间

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(!prices.size())return 0;
        int s[3]={ 0,-prices[0],0 };
        int ans=0;
        for(int i=1;i<prices.size();i++){
            int ori_s2=s[2];
            s[2]=s[1]+prices[i];
            s[1]=max(s[1],s[0]-prices[i]);
            s[0]=max(s[0],ori_s2);
            ans=max(ans,s[2]);
            ans=max(ans,s[0]);
        }
        return ans;
    }
};
```

版本(6

给定一个整数数组 `prices`，其中第 `i` 个元素代表了第 `i` 天的股票价格；非负整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每次交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

示例 1:

```
输入: prices = [1, 3, 2, 8, 4, 9], fee = 2
输出: 8
解释: 能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

注意:

- `0 < prices.length <= 50000`.
- `0 < prices[i] < 50000`.
- `0 <= fee < 50000`.

和版本2其实是一样的QAQ

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