## 【买股票的最佳时机】系列问题

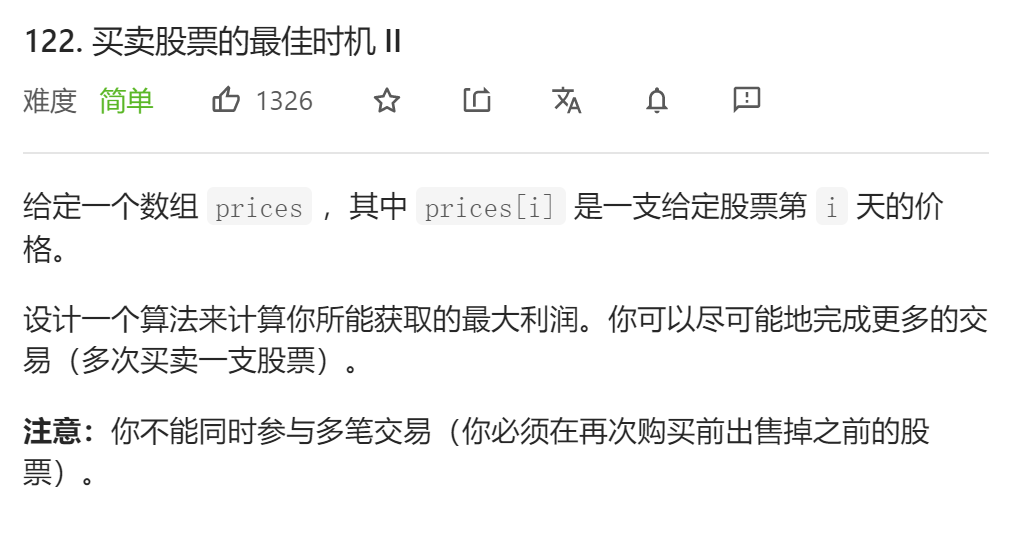
## 121.买股票的最佳时机

## fig:

1. **dp[i]表示在第i天卖出该股票的最大收益。**
2. **我们需要维护一个在第i天前股票的最低价的值**

class Solution {  
public:  
 int maxProfit(vector<int>& prices) {  
 int inf = 1e9;  
 int minprice = inf, maxprofit = 0;  
 for (int price: prices) {  
 maxprofit = max(maxprofit, price - minprice);  
 minprice = min(price, minprice);  
 }  
 return maxprofit;  
 }  
};  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/solution/121-mai-mai-gu-piao-de-zui-jia-shi-ji-by-leetcode-/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

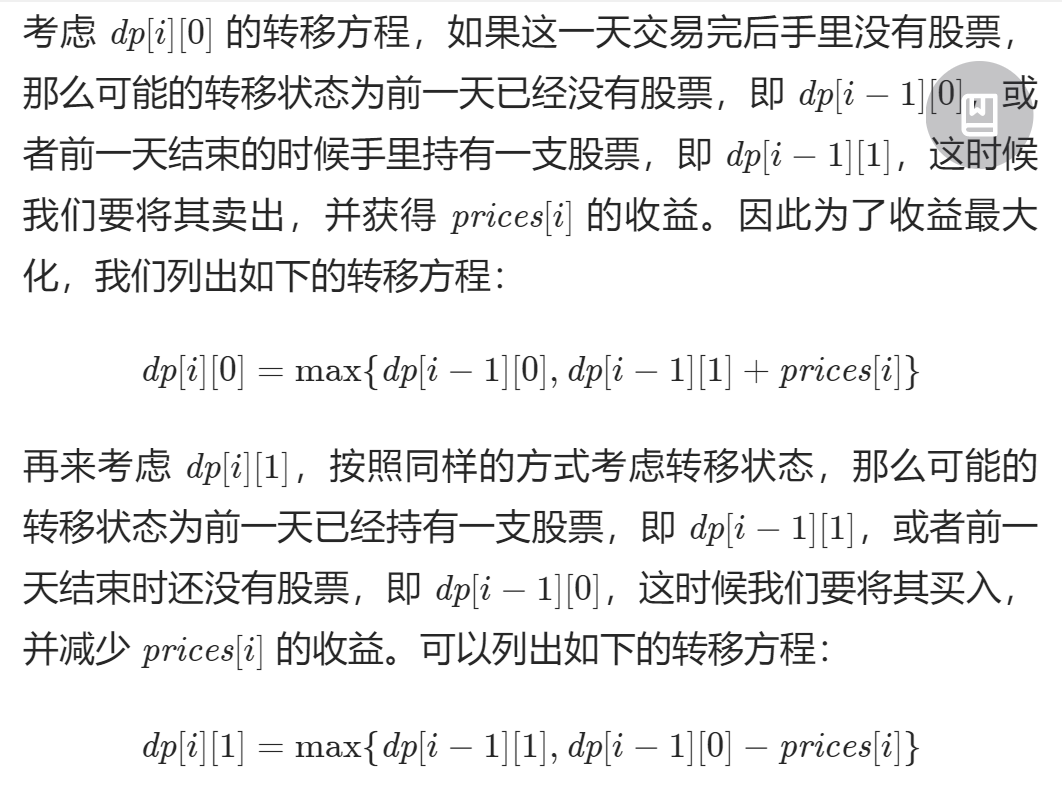
## 122.买卖股票的最佳实际Ⅱ



1. **贪心算法：找波峰和波谷；在波谷买入，波峰抛出。**

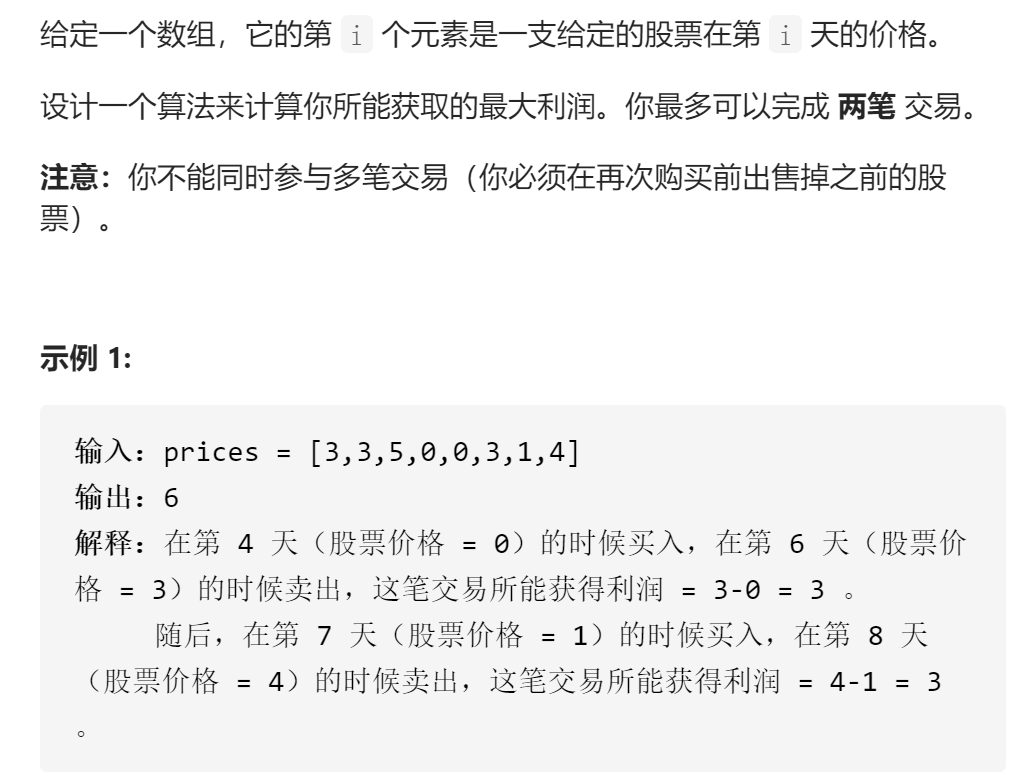
class Solution {  
public:  
 int maxProfit(vector<int>& prices) {   
 int ans = 0;  
 int n = prices.size();  
 for (int i = 1; i < n; ++i) {  
 ans += max(0, prices[i] - prices[i - 1]);  
 }  
 return ans;  
 }  
};  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/solution/mai-mai-gu-piao-de-zui-jia-shi-ji-ii-by-leetcode-s/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

1. **动态规划:**
2. **一个变量是时间，即天数，组成一个维度；**
3. **另一个维度是每一天的状态，我们可以对每一天的状态进行一个编码；0表示持有股票，1表示不持有股票；**
4. **dp[i] [j]表示第 i 天持有或不持有股票的最大收益**
5. **状态转移：**



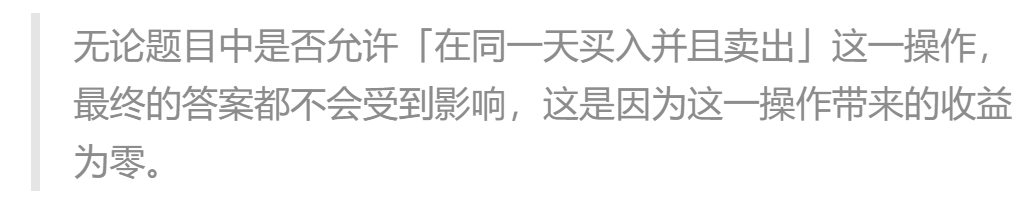
class Solution {  
public:  
 int maxProfit(vector<int>& prices) {  
 int n = prices.size();  
 int dp[n][2];  
 dp[0][0] = 0, dp[0][1] = -prices[0];  
 for (int i = 1; i < n; ++i) {  
 dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);  
 dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i]);  
 }  
 return dp[n - 1][0];  
 }  
};  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/solution/mai-mai-gu-piao-de-zui-jia-shi-ji-ii-by-leetcode-s/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 123：买卖股票的最佳时机Ⅲ



## 三维动态规划：

**关于初始化的一点说明：**

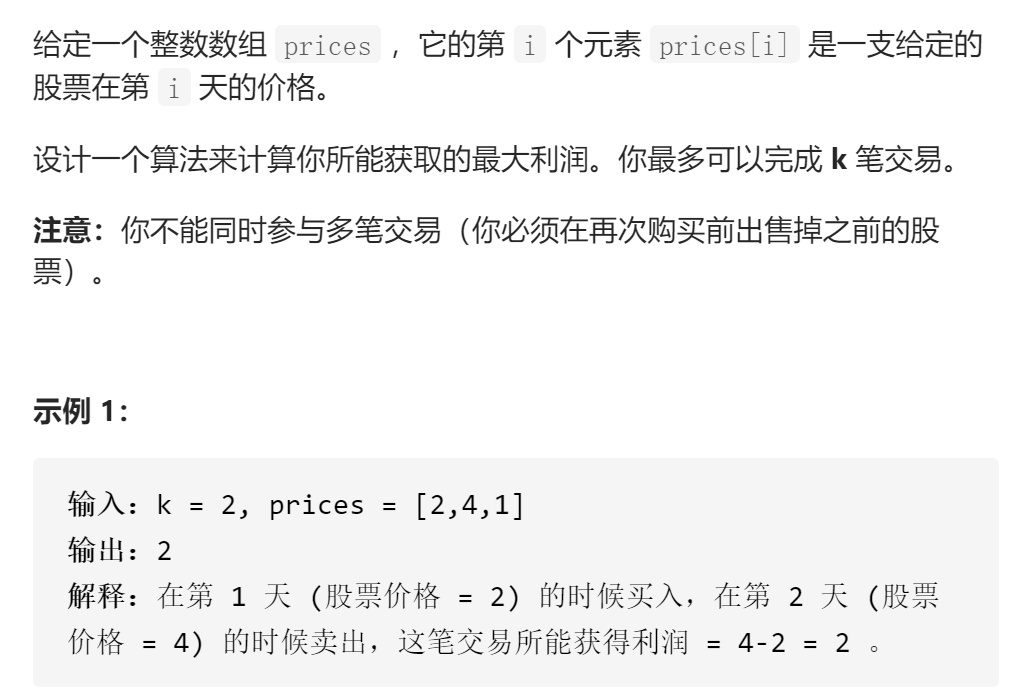


class Solution {  
public:  
 int maxProfit(vector<int>& prices) {  
 //dp[i][j][k]表示第i天持有或不持有；k为奇数时表示买入，为偶数时表示卖出  
 //这样的一个操作并不能完整的描述各个状态之间的关系  
 //尤其是不能很好的描述买入卖出之间的关系  
 int n=prices.size();  
 vector<vector<vector<int>>>dp(n,vector<vector<int>>(2,vector<int>(5,0)));  
 //注意初始化的方式，比较的特殊  
 dp[0][0][1]=-prices[0];  
 dp[0][0][3]=-prices[0];  
 for(int i=1;i<n;i++){  
 for(int j=0;j<2;j++){  
 for(int k=0;k<=4;k++){  
 if(j==0){  
 dp[i][j][k]=dp[i-1][0][k];  
 if(k>=1)  
 dp[i][j][k]=max(dp[i][j][k],dp[i-1][1][k-1]-prices[i]);  
 }else{  
 dp[i][j][k]=dp[i-1][1][k];  
 if(k>=1){  
 dp[i][j][k]=max(dp[i][j][k],dp[i-1][0][k-1]+prices[i]);  
 }  
 }  
 }  
 }  
 }  
 return max(dp[n-1][1][2],dp[n-1][1][4]);  
 }  
};

## 将买入和卖出分开的动态规划：

class Solution {  
public:  
 int maxProfit(vector<int>& prices) {  
 int n = prices.size();  
 int buy1 = -prices[0], sell1 = 0;  
 int buy2 = -prices[0], sell2 = 0;  
 for (int i = 1; i < n; ++i) {  
 buy1 = max(buy1, -prices[i]);  
 sell1 = max(sell1, buy1 + prices[i]);  
 buy2 = max(buy2, sell1 - prices[i]);  
 sell2 = max(sell2, buy2 + prices[i]);  
 }  
 return sell2;  
 }  
};  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iii/solution/mai-mai-gu-piao-de-zui-jia-shi-ji-iii-by-wrnt/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

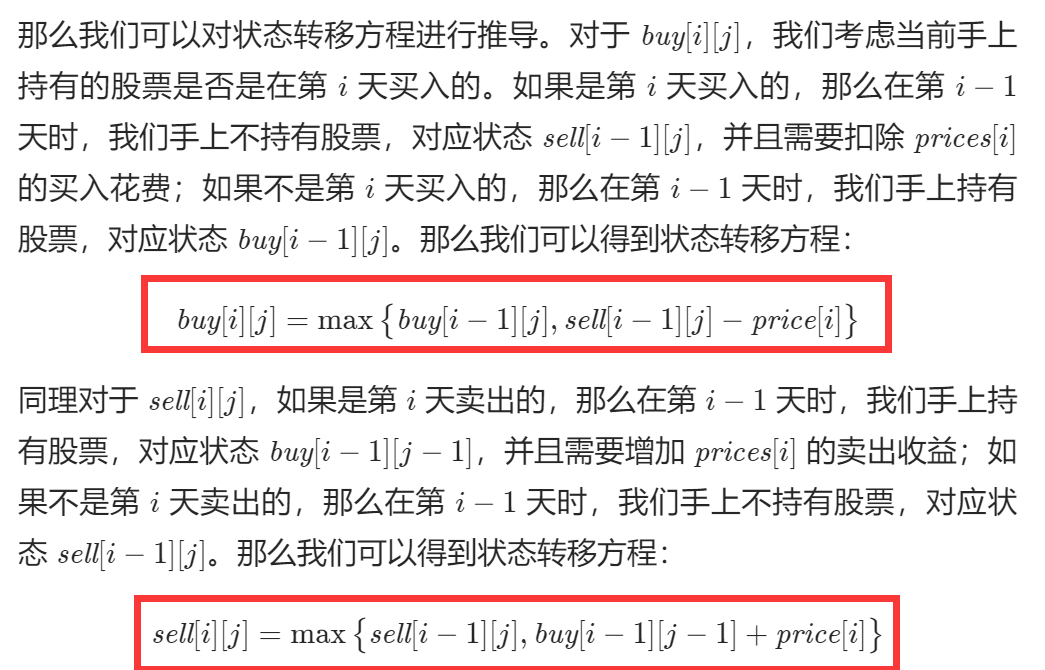
## 188：买卖股票的最佳时机Ⅳ



## 三维动态规划

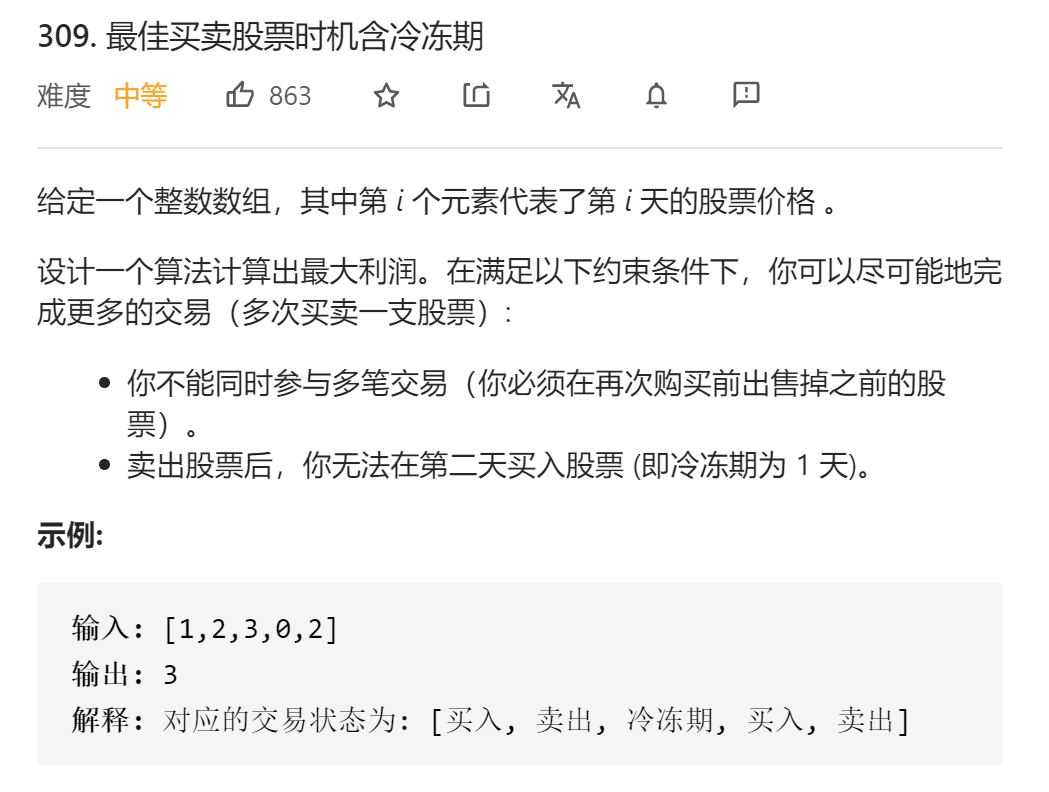
class Solution {  
public:  
 int maxProfit(int m, vector<int>& prices) {  
 //dp[i][j][k]表示第i天持有或不持有，已经进行k此抛售后的最大收益  
 //这样的一个操作并不能完整的描述各个状态之间的关系  
 //尤其是不能很好的描述买入卖出之间的关系  
 int n=prices.size();  
 if(n<=1)return 0;  
 vector<vector<vector<int>>>dp(n,vector<vector<int>>(2,vector<int>(2\*m+1,0)));  
 for(int i=0;i<m;i++)dp[0][0][2\*i+1]=-prices[0];  
 for(int i=1;i<n;i++){  
 for(int j=0;j<2;j++){  
 for(int k=0;k<=2\*m;k++){  
 if(j==0){  
 dp[i][j][k]=dp[i-1][0][k];  
 if(k>=1)  
 dp[i][j][k]=max(dp[i][j][k],dp[i-1][1][k-1]-prices[i]);  
 }else{  
 dp[i][j][k]=dp[i-1][1][k];  
 if(k>=1){  
 dp[i][j][k]=max(dp[i][j][k],dp[i-1][0][k-1]+prices[i]);  
 }  
 }  
 }  
 }  
 }  
  
 int res=0;  
 for(int i=0;i<m;i++)res=max(res,dp[n-1][1][2\*m]);  
 return res;  
 }  
};

## 将买入和卖出分开的动态规划



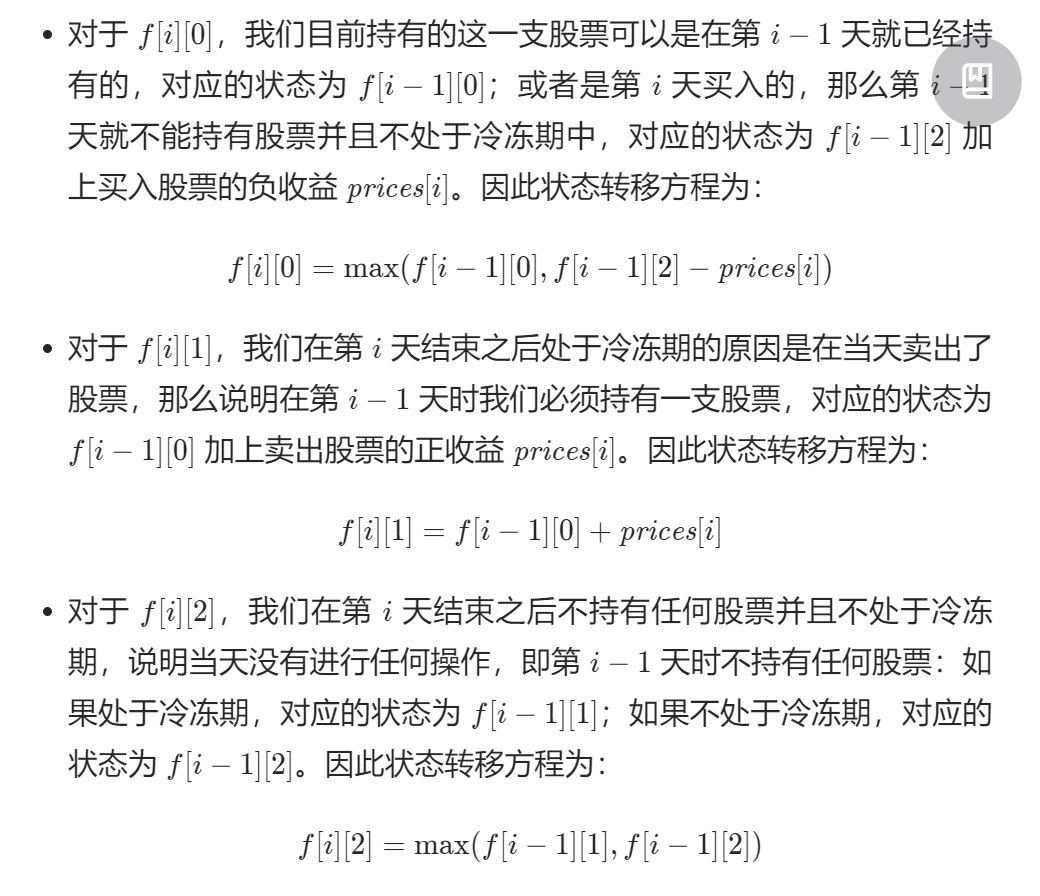
class Solution {  
public:  
 int maxProfit(int k, vector<int>& prices) {  
 if (prices.empty()) {  
 return 0;  
 }  
  
 int n = prices.size();  
 k = min(k, n / 2);  
 vector<vector<int>> buy(n, vector<int>(k + 1));  
 vector<vector<int>> sell(n, vector<int>(k + 1));  
  
 buy[0][0] = -prices[0];  
 sell[0][0] = 0;  
 for (int i = 1; i <= k; ++i) {  
 buy[0][i] = sell[0][i] = INT\_MIN / 2;  
 }  
  
 for (int i = 1; i < n; ++i) {  
 buy[i][0] = max(buy[i - 1][0], sell[i - 1][0] - prices[i]);  
 for (int j = 1; j <= k; ++j) {  
 buy[i][j] = max(buy[i - 1][j], sell[i - 1][j] - prices[i]);  
 sell[i][j] = max(sell[i - 1][j], buy[i - 1][j - 1] + prices[i]);   
 }  
 }  
  
 return \*max\_element(sell[n - 1].begin(), sell[n - 1].end());  
 }  
};  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/solution/mai-mai-gu-piao-de-zui-jia-shi-ji-iv-by-8xtkp/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 309:最佳买卖股票时机含冷冻期



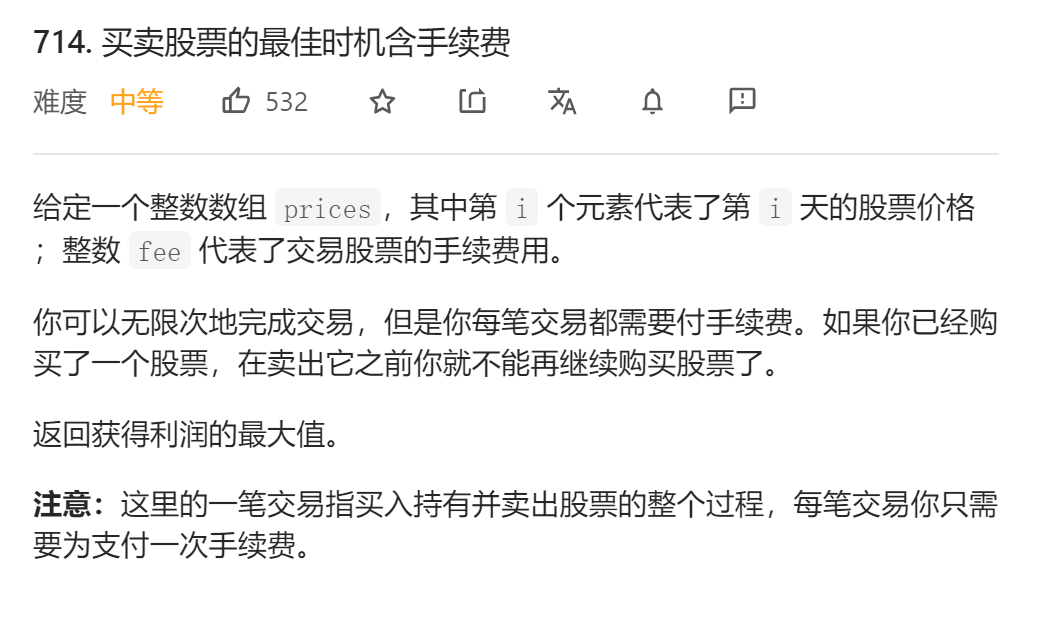
## 动态规划：

1. **第一变量是时间，即天数；作为一个维度**
2. **第二个变量是每天的一个状态，用数字进行一个编码。0表示持有股票；1表示处于冷冻期，不持有股票；2表示不持有股票，且不处于冷冻期，是一个可以自由买入的状态**
3. **状态转移：**



class Solution {  
public:  
 int maxProfit(vector<int>& prices) {  
 if (prices.empty()) {  
 return 0;  
 }  
  
 int n = prices.size();  
 // f[i][0]: 手上持有股票的最大收益  
 // f[i][1]: 手上不持有股票，并且处于冷冻期中的累计最大收益  
 // f[i][2]: 手上不持有股票，并且不在冷冻期中的累计最大收益  
 vector<vector<int>> f(n, vector<int>(3));  
 f[0][0] = -prices[0];  
 for (int i = 1; i < n; ++i) {  
 f[i][0] = max(f[i - 1][0], f[i - 1][2] - prices[i]);  
 f[i][1] = f[i - 1][0] + prices[i];  
 f[i][2] = max(f[i - 1][1], f[i - 1][2]);  
 }  
 return max(f[n - 1][1], f[n - 1][2]);  
 }  
};

## 714：买卖股票的最佳时机含手续费



class Solution {  
public:  
 int maxProfit(vector<int>& prices, int fee) {  
 //动态规划  
 //dp[i][0]表示第i天结束后手上持有股票的最大收益  
 //dp[i][1]表示第i天结束后受伤不持有股票的最大收益  
 int n=prices.size();  
 vector<vector<int>>dp(n,vector<int>(2,0));  
 //初始化  
 dp[0][0]=-prices[0];  
 for(int i=1;i<n;i++){   
 dp[i][0]=max(dp[i-1][0],dp[i-1][1]-prices[i]);   
 dp[i][1]=max(dp[i-1][1],dp[i-1][0]+prices[i]-fee);  
  
 }  
 return dp[n-1][1];  
 }  
};