* 买卖股票的最佳时机

1. 题目描述

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一次），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。（LeetCode 121.）

1. 题目分析

遇到小的就保存小的，遇到比小的大的就计算差值，保存最大差值。

1. 代码实现
2. **class** Solution {
3. **public**:
4. // 7 3 5 1 6 4
5. **int** maxProfit(vector<**int**>& prices) {
6. **int** n = prices.size();
7. **if**(!n) **return** 0;
8. **int** pre = prices[0];
9. **int** res = 0;
10. **for**(**int** i = 1; i < n; i++){
11. **if**(prices[i] < pre)
12. pre = prices[i];
13. **else**
14. res = max(res, prices[i]-pre);
15. }
17. **return** res;
18. }
19. };

* 买卖股票的最佳时机Ⅱ

1. 题目描述

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）。（LeetCode 122.）

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）。

1. 题目分析

两种贪心思路：

1. 当我比你大时，两种情况需要卖出：我是最后一个位置，或者我的后面的一个元素比我小，此时卖出
2. 只要当前位置比前面一个大就交易一次
3. 代码实现
4. 贪心
5. **class** Solution {
6. **public**:
7. **int** maxProfit(vector<**int**>& prices) {
8. **int** n = prices.size();
9. **if**(!n) **return** 0;
10. **int** res = 0;
11. **int** pre = prices[0];
13. **for**(**int** i = 1; i < n; i++){
14. **if**(prices[i] > prices[i-1]){
15. res += prices[i] - prices[i-1];
16. }
17. }
19. **return** res;
20. }
21. };
22. 动态规划
23. **class** Solution {
24. **public**:
25. //状态：天数，交易次数和有无股票，但是因为交易次数是无限多次，所以可以忽略
26. //状态转移方程：
27. //   第i天不持有股票的可能有两种：第i-1天就不持有，第i-1天持有第i天卖了，所有第i不持有股票的最大收益等于这两的最大值
28. //   dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
29. //   第i天持有股票的可能有两种：第i-1天就持有，第i-1天不持有第i天买了，所有第i持有股票的最大收益等于这两的最大值
30. //   dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
31. **int** maxProfit(vector<**int**>& prices) {
32. **int** n = prices.size();
33. **if**(!n) **return** 0;
34. **int** dp[n][2];
35. dp[0][0] = 0;   //第1天不持有股票的最大收益等于利润原点0
36. dp[0][1] = -prices[0];  //第1天持有股票的最大收益等于利润原点减去第一天的股票价格
37. **for**(**int** i = 1; i < n; i++){
38. dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);
39. dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i]);
40. }
42. **return** dp[n-1][0];  //最后一天不持有股票的收益肯定比最后一天持有股票的收益大
43. }
44. };

* 最佳买卖股票时机含冷冻期

1. 题目描述

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 。​（LeetCode 309.）

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)。

1. 题目分析

此题是在买卖股票的最佳时机Ⅱ的基础上加入了冷冻期的要求，所以大体框架还是买卖Ⅱ的框架，只需在一个细节上做出改动：

dp[i][1] = max(dp[i-1][1], dp**[i-2]**[0] – prices[i])

注意，第i天有股票有两种情况：

（1）第i-1天就有，今天无需操作；

（2）之前没有股票，今天买入，由于有冷冻期的要求，今天要买入，那么昨天必须是冷冻期，所以它是由dp[i-2][0]转换来的。所以现在dp[i-2]要推到dp[i]，初始化必须0和1都初始化。

1. 代码实现
2. **class** Solution {
3. **public**:
4. //状态：天数，有无股票
5. //状态转移：
6. //  dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
7. //  dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
8. **int** maxProfit(vector<**int**>& prices) {
9. **int** n = prices.size();
10. **if**(n <= 1) **return** 0;
11. **int** dp[n][2];
12. //边界条件
13. dp[0][0] = 0;
14. dp[0][1] = -prices[0];
15. dp[1][0] = max(dp[0][0], dp[0][1] + prices[1]);
16. dp[1][1] = max(dp[0][1], dp[0][0] - prices[1]);
17. **for**(**int** i = 2; i < n; i++){
18. dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);
19. dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i]);
20. }
22. **return** dp[n-1][0];
23. }
24. };

* 最佳买卖股票时机含手续费

1. 题目描述

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。（LeetCode 714.）

你可以无限次地完成交易，但是你每笔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。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

1. 题目分析

在买卖Ⅱ的基础上只是在i-1天卖出股票转到i天没有股票的状态时，利润还需要交出手续费。

1. 代码实现
2. **class** Solution {
3. **public**:
4. //状态：天数和有无股票
5. //状态转移：
6. // dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i] - fee)
7. // dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
8. **int** maxProfit(vector<**int**>& prices, **int** fee) {
9. **int** n = prices.size();
10. **if**(n <= 1) **return** 0;
11. **int** dp[n][2];
12. //边界状态
13. dp[0][0] = 0;
14. dp[0][1] = -prices[0];
15. **for**(**int** i = 1; i < n; i++){
16. dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i] - fee);
17. dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i]);
18. }
20. **return** dp[n-1][0];
21. }
22. };

* 买卖股票的最佳时Ⅲ

1. 题目描述

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。（LeetCode 123.）

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）。

1. 题目分析

贪心思想：将整个数据分为两段，前面一段进行一次交易，后面一段进行一次交易。

所以从前往后计算开始到每个位置所能最大的收益，从后往前计算每个位置到末尾所能的最大收益，然后加和求最大即可。

1. 代码实现
2. 贪心
3. **class** Solution {
4. **public**:
5. // [1,2,4,2,5,7,2,4,9,0]  是 1-7买入卖出，2-9买入卖出
6. **int** maxProfit(vector<**int**>& prices) {
7. **int** n = prices.size();
8. **if**(!n) **return** n;
9. vector<**int**> front(n, 0);
10. vector<**int**> back(n, 0);
12. **int** min1 = prices[0], max1 = prices[n-1];
13. **int** res1 = 0, res2 = 0;
14. **for**(**int** i = 0; i < n; i++){
15. **if**(prices[i] <= min1)
16. min1 = prices[i];
17. **else**
18. res1 = max(res1, prices[i] - min1);
19. front[i] = res1;
21. **if**(prices[n-i-1] >= max1)
22. max1 = prices[n-i-1];
23. **else**
24. res2 = max(res2, max1 - prices[n-i-1]);
25. back[n-i-1] = res2;
26. }
28. **int** res = 0;
29. **for**(**int** i = 0; i < n; i++){
30. res = max(res, front[i] + back[i]);
31. }
33. **return** res;
34. }
35. };
36. 动态规划
37. **class** Solution {
38. **public**:
39. // [1,2,4,2,5,7,2,4,9,0]  是 1-7买入卖出，2-9买入卖出
40. // 状态：天数，交易次数2和有无股票
41. // 状态转移方程
42. //  第i天进行第k次交易时不持有股票有两种可能：第i-1天就不持有股票，交易次数不需要增加；第i-1天持有股票，第i天卖出，买入视为多一次交易操作
43. //  dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
44. //  第i天进行第k次交易时持有股票有两种可能：第i-1天就持有股票，交易次数不需要增加；第i-1天不持有股票，第i天买入，则第i天多进行一次操作后操作次数为k次，那么第i-1天的操作次数应该是k-1次
45. //  dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
47. **int** maxProfit(vector<**int**>& prices) {
48. **int** n = prices.size();
49. **if**(!n) **return** n;
50. **int** dp[n][3][2];
51. //边界状态 i = 0 和 j = 0
52. **for**(**int** i = 0; i < n; i++){
53. dp[i][0][0] = 0;   // 第i天时交易了0次时不持有股票收益为0
54. dp[i][0][1] = INT\_MIN;  //第i天时交易了0次却持有股票是不可能的
55. }
56. **for**(**int** j = 1; j < 3; j++){
57. dp[0][j][0] = 0;    //第一天交易了j次不持有股票收益为0
58. dp[0][j][1] = -prices[0];   //第一天交易了j次持有股票收益为0-prices[0]
59. }
60. **for**(**int** i = 1; i < n; i++){
61. **for**(**int** k = 1; k <= 2; k++){
62. dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
63. dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
64. }
65. }
67. **return** dp[n-1][2][0];
68. }
69. };

* 买卖股票的最佳时间Ⅳ

买卖股票Ⅲ是此题的特殊情况

1. 题目描述

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。（LeetCode 188.）

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）。

1. 题目分析

动态规划实现，和买卖Ⅱ、买卖Ⅲ思路完全一致，买卖Ⅱ其实用贪心算法更好更快，但是动态规划更适合系列解法，因为交易次数无限所以省略到第二维交易次数，买卖Ⅲ就是买卖Ⅳ的代码把k换成2。买卖Ⅳ的特殊情况是当k>n/2时，最大交易次数相当于变成无限次交易（即相当于买卖Ⅱ）

1. 代码实现
2. **class** Solution {
3. **public**:
4. // 状态：天数，交易次数和有无股票
5. // 状态转移方程
6. //  第i天进行第j次交易时不持有股票有两种可能：第i-1天就不持有股票，交易次数不需要增加；第i-1天持有股票，第i天卖出，买入视为多一次交易操作
7. // dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices[i])
8. //  第i天进行第j次交易时持有股票有两种可能：第i-1天就持有股票，交易次数不需要增加；第i-1天不持有股票，第i天买入，则第i天多进行一次操作后操作次数为j次，那么第i-1天的操作次数应该是j-1次
9. // dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0] - prices[i])
10. **int** maxProfit(**int** k, vector<**int**>& prices) {
11. **if**(!k) **return** 0;
12. **int** n = prices.size();
13. **if**(!n) **return** 0;
14. **if**(k > n / 2) **return** maxProfit\_infinite(prices);
16. **int** dp[n][k+1][2];
17. //边界状态
18. //j = 0
19. **for**(**int** i = 0; i < n; i++){
20. dp[i][0][0] = 0; // 第i天时交易了0次时不持有股票收益为0
21. dp[i][0][1] = INT\_MIN;  //第i天时交易了0次却持有股票是不可能的
22. }
23. // i = 0
24. **for**(**int** j = 1; j <= k; j++){
25. dp[0][j][0] = 0;   //第一天交易了j次不持有股票收益为0
26. dp[0][j][1] = -prices[0];  //第一天交易了j次持有股票收益为0-prices[0]
27. }
29. **for**(**int** i = 1; i < n; i++){
30. **for**(**int** j = 1; j <= k; j++){
31. dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices[i]);
32. dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0] - prices[i]);
33. }
34. }
36. **return** dp[n-1][k][0];
38. }
40. **int** maxProfit\_infinite(vector<**int**>& prices) {
41. **int** n = prices.size();
42. **int** res = 0;
43. **int** pre = prices[0];
45. **for**(**int** i = 1; i < n; i++){
46. **if**(prices[i] > prices[i-1]){
47. res += prices[i] - prices[i-1];
48. }
49. }
51. **return** res;
52. }
53. };

* 系列文章[解读](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/solution/gu-piao-jiao-yi-xi-lie-cong-tan-xin-dao-dong-tai-g/)