股票利润问题及其三个拓展问题探究

原创 自动化所 邹嘉钰 LOA算法学习笔记 2021-01-05 10:36

01 问题描述

1.1 股票利润原问题

给定一个数组prices，数组的第i个元素代表股票第i天的价格。如果你**至多完成一次**交易（即买入和卖出股票一次），请设计一个算法来计算你能获取的最大利润。

1.2 股票利润拓展问题一

给定一个数组prices，数组的第i个元素代表股票第i天的价格。**不限制买卖次数**，请设计一个算法来计算你能获取的最大利润。注意，你不能同时参与多笔交易，也就是在购买股票前必须卖掉之前的股票。

1.3 股票利润拓展问题二

给定一个数组prices，数组的第i个元素代表股票第i天的价格。如果你**至多进行两次**交易，请设计一个算法来计算你能获取的最大利润。注意，你不能同时参与多笔交易，也就是在购买股票前必须卖掉之前的股票。

1.4 股票利润拓展问题三

给定一个数组prices，数组的第i个元素代表股票第i天的价格。如果你**至多进行k次**交易，请设计一个算法来计算你能获取的最大利润。注意，你不能同时参与多笔交易，也就是在购买股票前必须卖掉之前的股票。

02 股票利润原问题

2.1 问题结构分析

由于至多只能进行一次交易，存在两种情形：

1) 股票的价格一直在下跌，我们选择**一直不买不卖**，因为如果买入的话必然亏本。

2) 股票的价格有涨有跌，或一直在涨，存在盈利空间，于是我们想做出这样的选择：在不违背购买顺序（必须要购买股票，才能卖出股票）的前提下，**在股票价格最低的时候买入，股票价格最高的时候卖出**，这样就能获得最大的利润。

根据以上思路，可以写出动态规划算法，我们需要维护两个变量，变量low用来维护当前的股票最低价格，动态数组dp用来维护当前能获得的最大利润。其中n为股票价格数组的长度。

2.2 状态转移方程

$$low = min(prices[i], low)$$

$$dp[i] = max(dp[i-1], prices[i] - low)$$

2.3 算法正确性证明

用循环不变量予以证明。断言：截止到第i天为止，如果第i天的股票价格减去前(i-1)天中的最低股票价格大于上一轮的最大利润，那么就更新最大利润，否则保持不变。

- **初始化：**第一天，最低价格是就是第一天的股票价格，最大利润是0，因为仅有一天时间，如果买入的话利润为负，为了使利润最大，我们选择不买不卖。
- **循环：**设截止到第(i-1)天的最大利润是dp[i-1]，现在进入到第i天，如果第i天的价格prices[i]比上一轮的最低价格low更低，那么将low更新为prices[i]，否则保持不变；如果第i天的价格prices[i]与当前轮的最低价格low的差值比上一轮的最大理论dp[i-1]大， 那么更新当前轮的最大利润dp[i]为prices[i]-low，从而保证了每一轮总能取到最大利润。
- **终止：**从第1天开始往后一直比较到第n天，dp数组的最后一个值就是截止到最后一天为止的最大利润。

2.4 代码实现

```
def maxprofit(self,prices:List[int]):
    n = len(prices)
    if n<=1:
        return 0
    low = prices[0]
    dp = [0]*n
    for i in range(1,n):
        low = min(prices[i],low)
        dp[i] = max(dp[i-1],prices[i]-low)
    return dp[-1]
```

时间复杂度：一共有n次for循环，循环里面比较常数次（两次），因此时间复杂度为T(n) = O(n)。

03 股票利润拓展问题一

3.1 问题结构分析

最大利润的表达式为：

$$res = \sum_{i} prices[right] - prices[left]$$

由于不限制买卖次数，不失一般性，上述表达式的选取的两个价格区间端点left和right可以等价成所有长度为1的子区间（也就是不同的日期）的子集，因此：

$$res = \sum_{i=1}^{n-1} \max(0, prices[i] - prices[i-1])$$

我们采用贪心策略，如果第i天的股票价格大于前一天的股票价格，那么说明存在利润空间。只要“有利可图”，那么我们就不放过这部分子利润。将所有的子利润累加起来就是最终的利润。

3.2 算法正确性证明

用反证法，假设存在更优的买卖方案与贪心策略不同，不失一般性，假设prices[i]-prices[i-1]>0，第i天的子利润大于0，“更优”的买卖方案与贪心策略违背，在第(i-1)天或更早的时候就卖掉了股票，而贪心策略则会保住这部分大于0的子利润，比如在第(i-1)天买入股票、第i天卖出股票，赚到的利润为prices[i]-prices[i-1]，与新的“更优”方案矛盾，算法正确性得证。

3.3 代码实现

```python
def maxprofit(self,prices:List[int]):
    n = len(prices)
    if n<=1:
        return 0
    res = []
    for i in range(1,n):
        if prices[i]-prices[i-1]>0:
            res.append(prices[i]-prices[i-1])
    return sum(res) if res else 0
```

时间复杂度：n重for循环，每for循环里面仅比较一次，T(n) = O(n)。

04 股票利润拓展问题二

4.1 问题结构分析

这道变形题只限制进行至多两次买卖，思路与第一题很像，不同之处在于买卖次数最多可以为2。我们可以用动态规划的思想，维护一个三维的动态数组dp来记录当前的最大利润。其中dp[ ][ ][ ]的三个维度分别是天数、当前手上是否持有股票（如果持有，值为1，否则为0）、卖出的次数（至多为2）。

4.2 状态转移方程

1) 手上没股票，未卖出过，从未进行过交易

$$dp[i][0][0] = 0$$

2) 手上没股票，卖出过1次，可能是今天卖的，可能是之前卖的

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

3) 手上没股票，卖出过2次，可能是今天卖的，可能是之前卖的

$$dp[i][0][2] = max(dp[i-1][1][1] + prices[i], dp[i-1][0][2])$$

4) 手上有股票，未卖出过，可能是今天买的，可能是之前买的

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

5) 手上有股票，卖出过1次，可能是今天买的，可能是之前买的

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

6) 手上有股票，卖出过2次，由于次数限制这种情况不存在，因此初值设为无穷小

$$dp[i][1][2] = float('-inf')$$

4.3 算法正确性证明

算法正确性证明类似原问题，结合上述6个可能的状态转移情形易证。

4.4 代码实现

```
def maxProfit(self, prices: List[int]) -> int:
    if not prices:
        return 0
    n=len(prices)
    dp=[ [[0,0,0],[0,0,0] ] for i in range(n) ]
    # 给定dp初始化条件
    # 第一天休息
    dp[0][0][0]=0
    # 第一天买入
    dp[0][1][0]=-prices[0]
    # 第一天不可能已经有卖出
    dp[0][0][1] = float('-inf')
    dp[0][0][2] = float('-inf')
    #第一天不可能已经卖出
    dp[0][1][1]=float('-inf')
    dp[0][1][2]=float('-inf')
    for i in range(1,n):
        dp[i][0][0]=0
        dp[i][0][1]=max(dp[i-1][1][0]+prices[i],dp[i-1][0][1])
        dp[i][0][2]=max(dp[i-1][1][1]+prices[i],dp[i-1][0][2])
        dp[i][1][0]=max(dp[i-1][0][0]-prices[i],dp[i-1][1][0])
        dp[i][1][1]=max(dp[i-1][0][1]-prices[i],dp[i-1][1][1])
        dp[i][1][2]=float('-inf')
    return max(dp[n-1][0][1],dp[n-1][0][2],0)
```

时间复杂度：n次for循环，每次for循环里面比较常数次（6次），算法复杂度为T(n)=O(n)。

05 股票利润拓展问题三

5.1 问题结构分析

在上一题的基础上，限制买卖的次数从2变成了n,解决方案的大体思路是一样的。考虑动态规划算法，维护一个动态的三维数组dp，其中dp[ ][ ][ ]的三个维度分别是天数、卖出的次数（至多为k）、当前手上是否持有股票（如果持有，值为1，否则为0）。

5.2 状态转移方程

1) 如果第j天卖出，那么收益增加prices[i]；如果不卖，那么维持不变

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

2) 如果第j天买入，那么收益减少prices[i]；如果不买，那么维持不变

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

5.3 算法正确性证明

算法正确性证明类似拓展问题二，用循环不变量来证明。

5.4 代码实现

```
def maxProfit(self,k:int,prices:List[int]):
    n = len(prices)
    if n<= 1:
        return 0
    dp = [[[0] * 2 for j in range(k + 1)] for i in range(n)]
    for i in range(n):
 # 第一天不可能卖出股票，只能买入或不买
        dp[i][0][1] = -float('inf')
    for j in range(k + 1):
 # 如果买入，那么收益为-prices[0]
        dp[0][j][1] = -prices[0]
    for i in range(1, n):
        for j in range(1, k + 1):
            dp[i][j][0] = max(dp[i - 1][j][0], \
                              dp[i - 1][j][1] + prices[i])
            dp[i][j][1] = max(dp[i - 1][j][1], \
                              dp[i - 1][j - 1][0] - prices[i])
    return dp[-1][-1][0]
```

时间复杂度：外层for循环有n次，内层for循环有k次，内部比较两次，算法复杂度为T(n) = O(nk)=O(n)。

06 总结

后续三道拓展题目都是从第一道股票利润的题目中稍加改动和变形得到的,是一套挺有意思的题目，比较考验对动态规划和贪心的理解程度。

- 仔细分析问题的结构，找到问题的动态规划转移转移方程。

- 如果条件特殊一点，考虑用贪心。观察出了问题特殊性之后，将动态规划改成贪心，往往会简单一些，因为这利用到了问题本身的性质。
- 复杂一点的动态规划可能会用到三维或者更高维的dp数组。