缘起

之所以写这个答案是因为在看同学们的题解时，发现了一个奇怪的现象：有些同学的代码采用了k的倒序，而有些同学依然是正序。运行了两种代码后发现都可以正常AC，这就让我觉得疑惑。在动态规划入门的背包问题的解法中，很多人的文章说明了在何种转移方程下需要倒序。按照高赞回答的思路和状态转移方程，为了避免状态压缩造成的数据覆盖，此题应该采用k倒序才对，为何正序也可AC？为了解答这种疑惑，所以写了这篇文章。如果有同学跟我有同样的疑惑，不妨一看。

先说结论，k正序和倒序是不会影响最终结果的（废话，都AC了）。然而本文根据转移方程推导，证明了两种算法在本质上是一致的。

使用不同的转移方程可能会有完全不同的思路，导致不会出现我的疑惑。所以首先声明本文基于以下的转移方程，是借鉴高赞回答的。

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])

dp[i][j][0]表示最多买卖j次，第i天结束时空仓的最高收益。

dp[i][j][1]表示最多买卖j次，第i天结束时持仓的最高收益。

在该转移方程下，不知道为何需要倒序的同学，可以先看第一节——从转移方程出发，k倒序的必要性

如果明白为何要倒序，请直接看第二节——k正序的内在合理性分析

先贴一个k正序的解答：

public class Solution {

public int maxProfit(int k, int[] prices) {

//忽略前面

int[][] dp = new int[k][2];

for (int j = 0; j < k; j++) {

dp[j][1] = -9999;

}

for (int price : prices) {

for (int j = 0; j < k ; j++) {

if (j == 0) {

dp[j][1] = Math.max(dp[j][1], -price);

} else {

// 基本状态转移方程 1

dp[j][1] = Math.max(dp[j][1], dp[j - 1][0] - price);

}

// 基本状态转移方程 2

dp[j][0] = Math.max(dp[j][0], dp[j][1] + price);

}

}

return dp[k - 1][0];

}

}

作者：liweiwei1419

链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/solution/dong-tai-gui-hua-by-liweiwei1419-4/

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。

大部分答案都是k正序的，好不容易找到一个倒序的：

class Solution {

public:

int maxProfit(int k, vector<int>& prices) {

//忽略前面

vector<vector<int> > dp(k + 1, vector<int>{0, INT\_MIN});

for (int i = 0; i < N; ++i) {

for (int j = k; j > 0; --j) {

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

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

}

}

return dp[k][0];

}

};

作者：da-li-wang

链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/solution/c-zhuang-tai-ya-suo-dong-tai-gui-hua-by-da-li-wa-4/

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。

对比两个答案，可见除了边界条件和k的循环顺序外，其他完全一致。

一、从转移方程出发，k倒序的必要性

然而我做这道题时也参考了高赞回答的思路。按照这种思路，我认为k是必须倒序的。为什么呢？先上转移方程：

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])

dp[i][j][0]表示最多买卖j次，第i天结束时空仓的最高收益。

dp[i][j][1]表示最多买卖j次，第i天结束时持仓的最高收益。

可见，第i天只依赖于i-1天的数据，所以可以进行状态压缩。然而，压缩时发现，dp[i][k][1]需要依赖于dp[i-1][k-1][0]。如果按照k正序来计算，如下代码（忽略边界条件）：

for i in (0,n)

for j in (0, k)

dp[j][0] = Math.max(dp[j][0], dp[j][1] + price);

dp[j][1] = Math.max(dp[j][1], dp[j - 1][0] - price);

则最后一行计算dp[j][1](dp[i][j][1])时所用的dp[j-1][0]，本应为dp[i-1][j-1][0]。然而由于采用了状态压缩，i不变，j的前一个循环刚刚计算了dp[i][j-1][0]赋值给了dp[j-1][0]，所以dp[j - 1][0]实际为dp[i][j-1][0]，与转移方程产生了偏差。

让我们拿默认测试用例来看一下, 进行状态压缩的k正序和不进行压缩的区别：

输入: [2,4,1], k = 2

输出: 2

为了避免边界条件的讨论，此处先把边界条件写掉，如下：

k = 0 k = 1 k=2

i = 0 dp[0][0] = 0, dp[0][0][0] = 0

dp[0][1] = 0, dp[0][0][1] = 0 dp[1][0] = 0, dp[0][1][0] = 0

dp[1][1] = -2, dp[0][1][1] = -2 dp[2][0] = 0, dp[0][2][0] = 0

dp[2][1] = -2, dp[0][2][1] = -2

i = 1 dp[0][0] = 0, dp[1][0][0] = 0

dp[0][1] = 0, dp[1][0][1] = 0

i = 2 dp[0][0] = 0, dp[2][0][0] = 0

dp[0][1] = 0, dp[2][0][1] = 0

然后i和k都从1开始循环，先计算i = 1, k = 1， p(i) = 4的情况：

k = 0 k = 1 k=2

i = 0 dp[0][0] = 0, dp[0][0][0] = 0

dp[0][1] = 0, dp[0][0][1] = 0 dp[1][0] = 0, dp[0][1][0] = 0

dp[1][1] = -2, dp[0][1][1] = -2 dp[2][0] = 0, dp[0][2][0] = 0

dp[2][1] = -2, dp[0][2][1] = -2

i = 1 dp[0][0] = 0, dp[1][0][0] = 0

dp[0][1] = 0, dp[1][0][1] = 0 dp[1][0] = 2, dp[1][1][0] = 2

dp[1][1] = -2, dp[1][1][1] = -2

i = 2 dp[0][0] = 0, dp[2][0][0] = 0

dp[0][1] = 0, dp[2][0][1] = 0

此时并没有产生差别，但可以看到dp[1][0]已经被新值覆盖了，再也无法得到i=0时dp[1][0]的值了。此时继续计算i = 1，k = 2的情况，套用代码得到：

//使用三维辅助数组

dp[1][2][0] = max(dp[0][2][0], dp[0][2][1] + 4) = max(0, -2 + 4)

dp[1][2][1] = max(dp[0][2][1], dp[0][1][0] - 4) = max(-2, 0 - 4)

//状态压缩，使用二维数组，k为正序

dp[2][0] = Math.max(dp[2][0], dp[2][1] + 4) = max(0, -2 + 4)

dp[2][1] = Math.max(dp[2][1], dp[1][0] - 4) = max(-2, 2 - 4)

可见两种法在加粗部分已经产生了微小的差别。因为使用k正序的二维数组的dp[1][0]本应用i = 0时dp[0][1][0]的值 0，结果变成了i = 1时dp[1][1][0]的值 2。正如前面提到的，在使用k正序的二维辅助空间时，所有的dp[i-1][j-1][0]都会被dp[i][j-1][0]覆盖，导致错误。所以正是为了避免值被覆盖的情况发生，使用k倒序才显得必要。

二、k正序的内在合理性分析

其实再计算一步就可以发现，虽然在过程中使用的值有微小差别，但计算完成的结果竟然是一样的！

//使用三维辅助数组

dp[1][2][0] = max(dp[0][2][0], dp[0][2][1] + 4) = max(0, -2 + 4) = 0

dp[1][2][1] = max(dp[0][2][1], dp[0][1][0] - 4) = max(-2, 0 - 4) = -2

//状态压缩，使用二维数组，k为正序

dp[2][0] = Math.max(dp[2][0], dp[2][1] + 4) = max(0, -2 + 4) = 0

dp[2][1] = Math.max(dp[2][1], dp[1][0] - 4) = max(-2, 2 - 4) = -2

这其实给了我们一点提示，虽然计算过程中的一个变量值不同，但这一个值的差异并不会影响最终结果？下面使用状态转移方程给出证明：

//正确的转移方程

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

dp[i][j][1] = max(dp[i-1][j][1], \*\*dp[i-1][j-1][0]\*\* - prices[i]) (公式2)

//进行k正序的状态压缩后实际的转移方程

dp'[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1] + prices[i]) (公式3)

dp'[i][j][1] = max(dp[i-1][j][1], \*\*dp[i][j-1][0]\*\* - prices[i]) (公式4)

证明目标

显然dp[i][j][0] == dp'[i][j][0]是成立的，因为公式一样。所以我们的目标是证明dp[i][j][1] == dp'[i][j][1]，即：

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

要使dp[i][j][1] == dp'[i][j][1]成立，证明以下两个条件有一个成立即可：

条件1 (dp[i-1][j][1] >= dp[i-1][j-1][0] - prices[i]) && (dp[i-1][j][1] >= dp[i][j-1][0] - prices[i]) //这样max()永远等于dp[i-1][j][1]

条件2 dp[i-1][j-1][0] == dp[i][j-1][0] //这两个值相等，问题就不存在了

所以目标即证明：

(条件1 || 条件2) == true

条件1分析

将条件1拆分为：

dp[i-1][j][1] >= dp[i-1][j-1][0] - prices[i] 条件1-1

dp[i-1][j][1] >= dp[i][j-1][0] - prices[i] 条件1-2

将条件1-2中的dp[i][j-1][0]再运用一次转移方程(公式1)，则条件1-2等价于

dp[i-1][j][1] >= max(dp[i-1][j-1][0], dp[i-1][j-1][1] + prices[i]) - prices[i] 条件1-3

分情况讨论max里的情况，若dp[i-1][j-1][0] >= dp[i-1][j-1][1] + prices[i]，条件1-3转换为条件1-1：

dp[i-1][j][1] >= dp[i-1][j-1][0] - prices[i] 条件1-3转换为条件1-1

若dp[i-1][j-1][0] < dp[i-1][j-1][1] + prices[i]，条件1-3转换为：

dp[i-1][j][1] >= dp[i-1][j-1][1] + prices[i] - prices[i] = dp[i-1][j-1][1] 条件1-4

条件1-4显然成立，因为dp[i-1][j][1] >= dp[i-1][j-1][1]显然成立

综上，条件1等价于条件1-1，即：

dp[i-1][j][1] >= dp[i-1][j-1][0] - prices[i] 条件1-1

条件2分析

条件2中将dp[i][j-1][0]运用一次转移方程(公式1),条件2转化为：

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

则若下列条件2-2成立，条件2-1自然成立。

dp[i-1][j-1][0] >= dp[i-1][j-1][1] + prices[i] 条件2-2

又因为

dp[i-1][j][1] >= dp[i-1][j-1][1]

所以，若下列条件2-3成立，则条件2自然成立

dp[i-1][j-1][0] >= dp[i-1][j][1] + prices[i] 条件2-3

将条件2-3移项稍作整理即可得

dp[i-1][j][1] <= dp[i-1][j-1][0] - prices[i] 条件2-4

综合条件1、2

综上所述，条件1等价于条件1-1；若条件2-4成立，则条件2成立。观察条件1-1和条件2-4：

dp[i-1][j][1] >= dp[i-1][j-1][0] - prices[i] 条件1-1

dp[i-1][j][1] <= dp[i-1][j-1][0] - prices[i] 条件2-4

显然，这两个条件必有一条为真，所以目标(条件1 || 条件2) == true得证。k正序的合理性也就得到了证明。

三、结论

经过上面的分析，我的疑惑已经得到了解答。采用k的正序或者倒序，虽然计算过程中会有微小的差异，然而并不影响最终结果，这一结论是可以用转移公式推导来证明的。

上文的证明过程可能会有瑕疵，仅仅是本人的一点思考，如果有同学发现了错误或者问题，欢迎评论指出，也希望能在和大家的交流中共同进步。