掉头发系列之动态规划面试题

今天目标：

1：完成股票买卖系列面试题

股票买卖系列

股票买卖系列共6道题，如下：

121. 买卖股票的最佳时机

122. 买卖股票的最佳时机 II

123. 买卖股票的最佳时机 III

188. 买卖股票的最佳时机 IV

309. 最佳买卖股票时机含冷冻期

714. 买卖股票的最佳时机含手续费

经典的动态规划求解股票买卖问题。

算法分析：

> 前期分析过程：
>
> 确定状态和选择，其实对于选择我们很容易去确定，在第`i`天我们可以选择下列三种之一：**买入，卖出，不交易；**但这6道题中基本都限制了，买之前必须得先卖出（第一次除外），卖出前必须买入，且不能同时参与多笔交易。

1次	无数次	2次	k次	冷冻期	手续费
121	122	123	188	309	714

prices= [7 , 1 , 5 , 3 , 6 , 4]

1：dp[i] 代表到第 i 天的最大收益maxProfit，转移的过程：dp[i] = MAX (dp[i-1] - prices[i] ， dp[i-1] + prices[i] , dp[i-1])

买 卖 不动

存在的问题：在买入之前必须先把手里的卖出去，卖出去前必须先买进来，故一个维度不够，需要一个新的维度代表手里是否有股票

2：dp[i][j], 其中j取值{0，1}，j=0代表手里没有股票，j=1代表手里有股票，同样dp[i][j] 代表到第 i 天的最大收益maxProfit

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

不动 卖

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

不动 买

存在的问题：买卖次数是受k限制的，即最多买k次， 故还要继续升维

3：dp[i][k][j]，i取值于{0,1......n-1}，k取值于{0,1,2.......maxK}，j取值于{0,1} ，代表了到第 i 天，交易次数不超过k的情况下获得的最大收益maxProfit

for i: 0...n-1
 for k: 0...maxK
  for j: 0...1

dp[i][k][0] = MAX (dp[i-1][k][0] , dp[i-1][k][1] +prices[i])

不动 卖出

在第i天交易次数不超过k的情况下我没有持有股票，有两种可能：
1：昨天就没有持有股票，且今天选择不交易，
2：昨天持有股票，但是今天选择卖出

dp[i][k][1] = MAX(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

不动 买入

在第i天交易次数不超过k的情况下我持有股票，有两种可能
1：昨天就持有股票，且今天选择不交易
2：昨天没持有股票，今天选择买入股票

1、确定状态参数和选择

此处状态参数有第 `i` 天，交易次数 `k`，是否持有股票 `j`，

其中 `i` 的选择范围 `{0,1,2,.....n-1}`，`k` 的选择范围：`{0,1,2,.....maxK}`，`j` 的选择范围：`{0,1}`

选择也很简单：不交易，买入，卖出，但每天的选择是受限制的。

2、定义dp数组

`dp[i][k][j]` 代表了在第 `i` 天，交易次数不超过 `k` 次，持有股票或不持有股票下的最大收益。

3、状态转移逻辑

通过之前的分析，状态转移方程可列举如下：

```
dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);

dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i])
```

而我们最终要返回的是就是：`dp[n-1][maxK][0]`，

为什么不是 `dp[n-1][maxK][1]`，因为 `j=1` 代表手里持有股票，`j=0` 代表手里不持有股票，很明细 `j=0` 收益会更大。

4、初始化状态

```
dp[-1][k][0] = dp[i][0][0] = 0;
dp[-1][k][1] = dp[i][0][1] = -INFINITY
```

注意：

因为状态转移方程中有一个`i-1`，而`i`是从`0`开始的，故会产生`-1`的情况，至于怎么来表达`-1`的情况，方式有很多种

`-INFINITY`代表了负无穷，也就是一种不可能的情况。

有了这些相关思路有，我们可以依次去解决股票买卖的几个问题了

121. 买卖股票的最佳时机

该题中`maxK=1`，拿状态方程看一下

```
dp[i][1][0]=MAX(dp[i-1][1][0],dp[i-1][1][1] + prices[i]);

dp[i][1][1]=MAX(dp[i-1][1][1], dp[i-1][0][0] - prices[i]) =MAX( dp[i-1][1][1]
,  - prices[i])
```

解释一下：`k=0由初始条件dp[i-1][0][0]=0`

而且此时会发现，k都是为1，即k对状态转移没有影响了，因此可以简化掉k，变成如下

```
dp[i][0]=MAX(dp[i-1][0], dp[i-1][1] + prices[i]);
dp[i][1]=MAX( dp[i-1][1] ,  - prices[i])
```

代码实现如下：

```
class Solution {
    public int maxProfit(int[] prices) {
        //特殊
        if (prices==null || prices.length ==0) {
            return 0;
        }
        //定义dp
        int n = prices.length;
        //第i天持有或不持有股票的情况下的最大收益
        int[][] dp = new int[n][2];

        //转移
        for (int i=0;i<n;i++) {
             //针对边界初始情况处理一下
            if (i==0) { // 状态转移方程中需要计算dp[-1],我们单独计算即可
                /*
                    Math.max(dp[-1][0], dp[-1][1] + prices[i])
                    Math.max(0, -INFINITY + prices[i]) = 0;
                */
                dp[0][0] = 0;//第一天不持有股票,前面也没什么操作
                /*
                    Math.max(dp[-1][1], dp[-1][0] - prices[i])
                    Math.max(-INFINITY, 0 - prices[i]) = - prices[i]
                */
                dp[0][1] =  - prices[i];//第一天就买入股票,前面什么也没操作
```

```java

                    continue;
                }

                dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
                dp[i][1] = Math.max(dp[i-1][1],  - prices[i]);
            }

            return dp[n-1][0];
        }
    }
```

但是这样处理 初始条件 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，其实不用整个 dp 数组，只需要两个变量储存所需的状态就足够了，这样可以把空间复杂度降到 O(1)

```java
class Solution {
    public int maxProfit(int[] prices) {
        //特殊
        if (prices==null || prices.length ==0) {
            return 0;
        }
        //定义dp
        int n = prices.length;

        int dp_i_0 = 0;//初始值,对应dp[-1][0] = 0
        int dp_i_1 = Integer.MIN_VALUE;//初始值,对应dp[-1][1] = -INFINITY
        //转移
        for (int i=0;i<n;i++) {
            dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
            dp_i_1 = Math.max(dp_i_1,  - prices[i]);
        }

        return dp_i_0;
    }
}
```

122. 买卖股票的最佳时机 II

该题目中，买卖次数不受限制，可以认为`maxK=+INFINITY`，也可以认为`maxK=-INFINITY`，故也就证明`状态k`对状态转移不会产生影响，`k`和`k - 1`是一样的。

```
dp[i][k][0]=MAX(dp[i-1][k][0],dp[i-1][k][1] + prices[i]);

dp[i][k][1]=MAX(dp[i-1][k][1],dp[i-1][k-1][0]-prices[i])=MAX(dp[i-1][k][1],dp[i-1][k][0]- prices[i])
```

既然`k`对状态转移不会产生影响，那么可以去掉状态`k`编程如下：

```
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])
```

同样，对于初始化条件

```
dp[-1][0] = 0;
dp[-1][1] = -INFINITY
```

直接代码实现：

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int dp_i_0 = 0;
        int dp_i_1 = Integer.MIN_VALUE;
        for (int i=0;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])
            */
            dp_i_0 = Math.max(dp_i_0,dp_i_1+prices[i]);
            dp_i_1 = Math.max(dp_i_1,dp_i_0-prices[i]);
        }
        return dp_i_0;
    }
}
```

309. 最佳买卖股票时机含冷冻期

该题目中，买卖次数不受限制，可以认为 `maxK=+INFINITY`，也可以认为 `maxK=-INFINITY`，故也就证明 `状态k` 对状态转移不会产生影响， `k` 和 `k - 1,k-2` 等等 是一样的。

另外，卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。此时我们的状态转移方程需要做一个变更

```
//通用的状态转移方程
dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i]);

//结合该题改造后的方程,该题的要求是第i天准备买的时候,前一天(i-1)不能卖，即前一天不能做任何操作，因此收益继承自前两天，也就是说第i天的状态只能从(i-2)天转移过来
dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-2][k-1][0] - prices[i]);

//又由于k对状态转移没有影响故去掉状态k
dp[i][0] = MAX ( dp[i-1][0] , dp[i-1][1] + prices[i]);
dp[i][1] = MAX ( dp[i-1][1] , dp[i-2][0] - prices[i]);
```

初始条件：

```
//原始
dp[-1][k][0] = dp[i][0][0] = 0;
dp[-1][k][1] = dp[i][0][1] = -INFINITY

//本题的初始化状态
dp[-1][0] = 0;
dp[-2][0] = 0;
dp[-1][1] = -INFINITY
```

直接代码实现为:

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int dp_i_0 = 0;
        int dp_i_2_0 = 0;
        int dp_i_1 = Integer.MIN_VALUE;
        for (int i=0;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-2][0] - prices[i]);
            */
            int temp = dp_i_0;
            dp_i_0 = Math.max(dp_i_0,dp_i_1+prices[i]);
            dp_i_1 = Math.max(dp_i_1,dp_i_2_0 - prices[i]);
            dp_i_2_0 = temp;
        }
        return dp_i_0;
    }
}
```

714. 买卖股票的最佳时机含手续费

该题目中，买卖次数不受限制，可以认为 `maxK=+INFINITY`，也可以认为 `maxK=-INFINITY`，故也就证明 `状态k` 对状态转移不会产生影响， `k` 和 `k - 1,k-2` 等等 是一样的，可从状态参数中去掉 `k`

另外，每次交易要支付手续费，只要把手续费从利润中减去即可，故我们定义在买入的时候将手续费扣除。

状态转移方程改造如下:

```
//原始状态转移方程
dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i])

//改造后的状态转移方程
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)
```

初始化状态:

```
dp[-1][0] = 0;
dp[-1][1] = -INFINITY
```

直接写出代码

```java
class Solution {
    public int maxProfit(int[] prices, int fee) {
        int n = prices.length;
        /*
            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)
        */
        int dp_i_0 = 0;
        int dp_i_1 = Integer.MIN_VALUE;

        for (int i=0;i<n;i++) {
            int temp = dp_i_0;
            dp_i_0 = Math.max(dp_i_0,dp_i_1 + prices[i] );
            dp_i_1 = Math.max(dp_i_1, temp - prices[i] -fee);
        }
        return dp_i_0;
    }
}
```

> 注意：手续费的扣除也可以选择在卖出的时候扣除，即
>
> ```
> dp[i][0] = MAX ( dp[i-1][0] , dp[i-1][1] + prices[i] - fee );
> dp[i][1] = MAX ( dp[i-1][1] , dp[i-1][0] - prices[i])
> ```
>
> 只不过代码编写的时候需要注意一下，初始条件当`i`等于0时，会产生`dp[-1][1]`的值，我们当时定的`dp[-1][1]=Integer.MIN_VALUE`就不能这样设置了，这个地方`-fee`后会导致`int`存储范围越界，不过我们可以改一下初始条件`dp[-1][1]=Integer.MIN_VALUE+fee`，代码如下
>
> ```java
> class Solution {
> public int maxProfit(int[] prices, int fee) {
>     int n = prices.length;
>     /*
>         dp[i][0] = MAX ( dp[i-1][0] , dp[i-1][1] + prices[i] - fee );
>         dp[i][1] = MAX ( dp[i-1][1] , dp[i-1][0] - prices[i] )
>     */
>     int dp_i_0 = 0;
>     int dp_i_1 = Integer.MIN_VALUE+fee;
>
>     for (int i=0;i<n;i++) {
>         int temp = dp_i_0;
>         dp_i_0 = Math.max(dp_i_0,dp_i_1 + prices[i] -fee);
>         dp_i_1 = Math.max(dp_i_1, temp - prices[i] );
>     }
>     return dp_i_0;
> }
> }
> ```
>
> 不过更好的做法是，我们遍历的时候从`i=1`开始遍历

```java
class Solution {
public int maxProfit(int[] prices, int fee) {
    int n = prices.length;
    /*
        dp[i][0] = MAX ( dp[i-1][0] , dp[i-1][1] + prices[i] - fee );
        dp[i][1] = MAX ( dp[i-1][1] , dp[i-1][0] - prices[i] )
    */
    int dp_i_0 = 0;
    int dp_i_1 = -prices[0];//注意初始条件,代表第一天买入

    for (int i=1;i<n;i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0,dp_i_1 + prices[i] -fee);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i] );
    }
    return dp_i_0;
}
}
```

123. 买卖股票的最佳时机 III

该题中，`maxK=2`，

```
//原始状态转移方程
dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i])
```

根据我们之前所给出的状态转移的通用模板来看

```
//进行状态转移
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)
```

我们现在需要遍历这三个状态：`i` 代表的第几天，`k` 代表的是最多交易次数，`j` 代表是否持有股票

但由于本题k取值的特殊性 `k=2` 因此可以像状态 `j` 一样进行枚举，反正 `k` 的取值要么是 `1` 要么是 `2`。

所以状态转移方程可改造为：

```
//k=1
dp[i][1][0] = MAX ( dp[i-1][1][0] , dp[i-1][1][1] + prices[i]);
dp[i][1][1] = MAX ( dp[i-1][1][1] , dp[i-1][1-1][0] - prices[i])
//k=2
dp[i][2][0] = MAX ( dp[i-1][2][0] , dp[i-1][2][1] + prices[i]);
dp[i][2][1] = MAX ( dp[i-1][2][1] , dp[i-1][2-1][0] - prices[i])
```

简化后为如下形式：

```
dp[i][1][0] = MAX ( dp[i-1][1][0] , dp[i-1][1][1] + prices[i]);
dp[i][1][1] = MAX ( dp[i-1][1][1] ,  - prices[i])   //dp[i-1][1-1][0] = 0
dp[i][2][0] = MAX ( dp[i-1][2][0] , dp[i-1][2][1] + prices[i]);
dp[i][2][1] = MAX ( dp[i-1][2][1] , dp[i-1][1][0] - prices[i])
```

初始化状态：

```
dp[-1][1][0] = 0;
dp[-1][1][1] = -INFINITY;
dp[-1][2][0] = 0;
dp[-1][2][1] = -INFINITY;
```

注意：最后返回的是 `dp[n-1][k][0]` 的值

故直接用状态压缩写出O(1)空间复杂度的代码来：

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        /*
            状态转移方程:
                dp[i][1][0] = MAX ( dp[i-1][1][0] , dp[i-1][1][1] + prices[i]);
                dp[i][1][1] = MAX ( dp[i-1][1][1] ,  - prices[i])   //dp[i-1][1-1][0] = 0

                dp[i][2][0] = MAX ( dp[i-1][2][0] , dp[i-1][2][1] + prices[i]);
                dp[i][2][1] = MAX ( dp[i-1][2][1] , dp[i-1][1][0] - prices[i]) //由于这里依赖dp[i-1][1][0],故先计算k=2的情况
            初始条件:
                dp[-1][1][0] = 0;
                dp[-1][1][1] = -INFINITY;
                dp[-1][2][0] = 0;
                dp[-1][2][1] = -INFINITY;
        */
        int i_1_0 =0;
        int i_1_1 = Integer.MIN_VALUE;
        int i_2_0 = 0;
        int i_2_1 = Integer.MIN_VALUE;
        for (int i=0;i<n;i++) {
            i_2_0 = Math.max(i_2_0,i_2_1 + prices[i]);
            i_2_1 = Math.max(i_2_1,i_1_0 - prices[i]);

            i_1_0 = Math.max(i_1_0,i_1_1 + prices[i]);
            i_1_1 = Math.max(i_1_1, -prices[i]);
        }

        return i_2_0;
    }
}
```

188. 买卖股票的最佳时机 IV

该题中，状态参数 `k` 可以是任意值，就不可能再像上一题那样枚举了，只能通过 `for` 循环的方式穷举。

状态方程：

```
//原始状态转移方程
dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i])
```

初始化状态

```
dp[-1][k][0] = dp[i][0][0] = 0;
dp[-1][k][1] = dp[i][0][1] = -INFINITY
```

根据状态转移方程和初始化状态，写出代码为：

```java
class Solution {
    public int maxProfit(int maxK, int[] prices) {
        //特殊
        if (prices==null || prices.length ==0) {
            return 0;
        }
        //定义maxK的有效范围
        maxK = Math.min(maxK,prices.length /2 );

        int n = prices.length;
        //定义dp
        int[][][] dp = new int[n][maxK+1][2];//第二维的长度是maxK+1的原因是:dp[i][k]代表的是第i天交易次数不超过k次,下标为k的数组长度是k+1
        /*
            状态转移方程:
                dp[i][k][0] = MAX ( dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
                dp[i][k][1] = MAX ( dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i])
            初始化状态:
                dp[-1][k][0] = dp[i][0][0] = 0;
                dp[-1][k][1] = dp[i][0][1] = -INFINITY
        */
        for (int i=0;i<n;i++) {
            for (int k=1;k<=maxK;k++) {
                if (i==0) {//处理初始化条件
                    dp[0][k][0] = 0;
                    dp[0][k][1] = -prices[0];
                    continue;
                }
                dp[i][k][0] = Math.max(dp[i-1][k][0] , dp[i-1][k][1] + prices[i]);
                dp[i][k][1] = Math.max(dp[i-1][k][1] , dp[i-1][k-1][0] - prices[i]);
            }
        }
```

```
        return dp[n-1][maxK][0];
    }
}
```

其他经典题目

- [53. 最大子序和]
- [152. 乘积最大子数组]
- [120. 三角形最小路径和]
- [10. 正则表达式匹配]
- [44. 通配符匹配]