动态规划-买卖股票的最佳时机 专题

买卖股票的最佳时机（可以用贪心）

[力扣题目链接]

因为只能买卖一次。

`dp[i][j]` 数组的含义：第 i 天状态为 j 时的最大现金。

递推公式：

设 `dp[i][0]` 为持有股票状态， `dp[i][1]` 为不持有股票状态。

当 该状态是**持有股票状态**：

- 如果当天还是保持**持有股票状态**，那么保持原状， `dp[i][0]` = `dp[i-1][0]` ;
- 如果当天选择**买入股票**，所得现金就是买入今天的股票后所得现金即：`-prices[i]`

那么选取最大的即可：`dp[i][0]` = Math.max(`dp[i-1][0]` , `-prices[i]`);

当 该状态是**不持有股票状态**：

- 如果当天还是保持不持有股票状态，那么保持原状， `dp[i][1]` = `dp[i-1][1]` ;
- 如果当天是卖出股票状态，那么最大现金就是 `dp[i][1]` = `dp[i-1][0]` + `prices[i]` (即 持有股票状态的最大现金 `dp[i-1][0]` + 今天的股票价格 `prices[i]`)

那么选取最大的即可：`dp[i][1]` = Math.max(`dp[i-1][1]` , `dp[i-1][0]` + `prices[i]`);

```
// 动归
public int maxProfit1(int[] prices) {
    int[][] dp = new int[prices.length][2];
    int result = 0;
    dp[0][0] = -prices[0];
    dp[0][1] = 0;
    for (int i = 1; i < prices.length; i++) {
          dp[i][0] = Math.max(dp[i - 1][0], -prices[i]);
          dp[i][1] = Math.max(dp[i - 1][1], prices[i] + dp[i - 1][0]);
      }
    return dp[dp.length - 1][1];
}
/*
时间复杂度：O(n)
空间复杂度：O(n)
*/

// 贪心
public int maxProfit(int[] prices) {
        // 找到一个最小的购入点
        int low = Integer.MAX_VALUE;
        // res不断更新，直到数组循环完毕
        int res = 0;
        for(int i = 0; i < prices.length; i++){
            low = Math.min(prices[i], low);
```

```java
            res = Math.max(prices[i] - low, res);
        }
        return res;
    }
/*
时间复杂度：O(n)
空间复杂度：O(1)
*/
```

买卖股票的最佳时机2（可以用贪心）

[力扣题目链接]()

可以多次买卖股票

因为这里的股票可以买卖多次了，那么和第一个的**唯一区别**就是在递推公式上

在持有股票的状态上：`dp[i][0]`

- 假如当天是买入股票，那么就是 `dp[i][0]` = `dp[i-1][1]` - `prices[i]` （即 不持有股票状态的最大现金 `dp[i][1]` - 当天的股票价格 `prices[i]`）

```java
// 动归
public int maxProfit(int[] prices) {
        int[][] dp = new int[prices.length][2];
        int result = 0;
        dp[0][0] = -prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < prices.length; i++) {
            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] + dp[i - 1][0]);
        }
        return dp[dp.length - 1][1];
    }
/*
时间复杂度：O(n)
空间复杂度：O(n)
*/

// 贪心
public int maxProfit(int[] prices) {
        int result = 0;
        for (int i = 1; i < prices.length; i++) {
            result += Math.max(prices[i] - prices[i - 1], 0);
        }
        return result;
}
/*
时间复杂度：O(n)
空间复杂度：O(1)
*/
```

买卖股票的最佳时机3

[力扣题目链接]

最多可以完成两笔交易， 意味着可以买卖一次，可以买卖两次，可以不买卖。

两笔交易，也就是最多四个状态。即 第一次买入，第一次卖出，第二次买入，第二次卖出

`dp[i][j]` 数组的含义：在第 i 天状态为 j 的时候的所得最大金钱。

递推公式：

在**第一次持有股票状态**下： `dp[i][1]`

- 保持持有股票的状态，即 `dp[i][1]` = `dp[i-1][1]`
- 第一次买入股票，即 `dp[i][1]` = `-prices[i]`

`dp[i][1]` = Math.max(`dp[i-1][1]` , `-prices[i]`);

在**第一次不持有股票状态**下： `dp[i][2]`

- 保持不持有股票的状态，即 `dp[i][2]` = `dp[i-1][2]`
- 第一次卖出股票， `dp[i][2]` = `dp[i-1][1] + prices[i]`

`dp[i][2]` = Math.max(`dp[i-1][1] + prices[i]` , `dp[i-1][2]`)

在**第二次持有股票状态**下： `dp[i][3]`

- 当天保持持有股票，即 `dp[i][3]` = `dp[i-1][3]`
- 当天买入股票，就要看**第一次不持有股票时**的最大金钱了，即 `dp[i][3]` = `dp[i-1][2]` - `prices[i]`

`dp[i][3]` = Math.max(`dp[i-1][2] - prices[i]` , `dp[i-1][3]`)

在**第二次不持有股票状态**下： `dp[i][4]`

- 保持不持有股票的状态，即 `dp[i][4]` = `dp[i-1][4]`
- 当天卖出股票，就需要看**第二次持有股票时**的最大金钱了，即 `dp[i][4]` = `dp[i-1][3]` + `prices[i]`

`dp[i][4]` = Math.max(`dp[i-1][3] + prices[i]` , `dp[i-1][4]`)

初始化（结合这个dp数组的含义去想）：

`dp[0][0]` = 0 （可以省略，因为上述都没有设置 **没有操作** 的这个选择）

`dp[0][1]` = -prices[0]

`dp[0][2]` = 0

`dp[0][3]` = -prices[0]

`dp[0][4]` = 0

```
public int maxProfit(int[] prices) {
        int len = prices.length;
        if (prices.length == 0) return 0;
        int[][] dp = new int[len][5];
        dp[0][1] = -prices[0];
        // 初始化第二次买入的状态是确保 最后结果是最多两次买卖的最大利润
```

```
        dp[0][3] = -prices[0];
        for (int i = 1; i < len; i++) {
            dp[i][1] = Math.max(dp[i - 1][1], -prices[i]);
            dp[i][2] = Math.max(dp[i - 1][2], dp[i][1] + prices[i]);
            dp[i][3] = Math.max(dp[i - 1][3], dp[i][2] - prices[i]);
            dp[i][4] = Math.max(dp[i - 1][4], dp[i][3] + prices[i]);
        }
        return dp[len - 1][4];
    }
/*
时间复杂度：O(n)
空间复杂度：O(n*5)
*/
```

买卖股票的最佳时机4

力扣题目链接

最多可以完成K笔交易， 是时机3的进阶版

`dp[i][j]` 数组含义依旧。

参考时机3的做法:

0：表示不操作;

1：表示第一次买入;

2：表示第一次卖出;

3：表示第二次买入;

4：表示第二次卖出;

。。。。 可以发现，**除了0以外，奇数次为买入股票，偶数次为卖出股票。**

题目要求的是至多 K 笔交易，j 的定义范围就是到了 2 * K + 1即可。即 `int[][] dp = new int[prices.length][2 * K + 1]`

递推公式:

奇数次：`dp[i][j+1] = Math.max(dp[i-1][j+1],dp[i-1][j] - prices[i])`

偶数次：`dp[i][j+2] = Math.max(dp[i-1][j+2],dp[i-1][j+1] + prices[i])`

```
public int maxProfit(int k, int[] prices) {
        if (k < 1 && prices.length == 0) return 0;
        int[][] dp = new int[prices.length][2 * k + 1];
        for (int i = 1; i < 2 * k; i += 2) {
            dp[0][i] = -prices[0];
        }
        for (int i = 1; i < prices.length; i++) {
            for (int j = 0; j < 2 * k - 1; j += 2) {
                dp[i][j + 1] = Math.max(dp[i - 1][j + 1], dp[i - 1][j] -
prices[i]);
                dp[i][j + 2] = Math.max(dp[i - 1][j + 2], dp[i - 1][j + 1] +
prices[i]);
```

```
            }
        }
        return dp[prices.length - 1][2 * k];
    }
/*
时间复杂度：O(n)
空间复杂度：O(n*k)
*/
```

买卖股票的最佳时机含冷冻期

力扣题目链接

可以进行多次的买卖股票，但是卖出股票的第二天不能有其他操作。

dp数组含义依旧。

递推公式：

状态一：当持有股票状态：保持原状（**前些天买的股票，今天维持**；或者是**当天买入股票**） `dp[i][0]`

当不持有股票状态：两种卖出股票状态

- 状态二：当天依旧是保持卖出股票状态（前两天卖出股票，度过一天冷冻期；前一天卖出股票状态，一直没操作） `dp[i][1]`
- 状态三：当天卖出股票 `dp[i][2]`

状态四：今天为冷冻期，不能操作股票，只有一天。 `dp[i][3]`

为什么在之前的做题中「今天卖出股票」没有单独列为一个状态，而不是归类于「今天不持有股票」状态?

因为冷冻期的前一天**一定是**「今天卖出股票」状态，而不能是「今天卖不持有股票」状态，毕竟「今天卖不持有股票」状态也包含了「今天卖出股票」状态，但是不一定是「今天卖出股票」。

所以递推公式为：

状态一： `dp[i][0]`

- 操作一：前一天就是持有股票状态，今日继续保持 `dp[i][0]` = `dp[i-1][0]`
- 操作二：今日买入了股票，有两种情况：
  - 前一天是冷冻期（状态四）： `dp[i-1][3]` - `prices[i]`
  - 前一天是保持卖出股票状态： `dp[i-1][1]` - `prices[i]`

`dp[i][0]` = Math.max(`dp[i-1][0]` ,Math.max(`dp[i-1][3]` - `prices[i]` , `dp[i-1][1]` - `prices[i]`))

状态二： `dp[i][1]`

- 前一天卖出股票状态，一直没操作（状态二） `dp[i][1]` = `dp[i-1][1]`
- 前两天卖出股票，度过一天冷冻期 `dp[i][1]` = `dp[i-1][3]`

`dp[i][1]` = Math.max(`dp[i-1][1]` , `dp[i-1][3]`)

状态三： `dp[i][2]`

- 前一天持有股票状态 `dp[i-1][0]` + `prices[i]`

`dp[i][2]` = `dp[i-1][0]` + `prices[i]`

状态四：`dp[i][3]`

`dp[i][3]` = `dp[i-1][2]`

```
public int maxProfit(int[] prices) {
        int n = prices.length;
         if (n == 0) return 0;
        int[][] dp = new int[n][4];
         dp[0][0] -= prices[0]; // 持股票
         for (int i = 1; i < n; i++) {
             dp[i][0] = Math.max(dp[i-1][0],Math.max(dp[i-1][3] - prices[i] ,
dp[i-1][1] - prices[i]));
             dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][3]);
             dp[i][2] = dp[i - 1][0] + prices[i];
             dp[i][3] = dp[i - 1][2];
         }
         return Math.max(dp[n - 1][3], Math.max(dp[n - 1][1], dp[n - 1][2]));
    }

/*
时间复杂度：O(n)
空间复杂度：O(n)
*/
```

买卖股票的最佳时机含手续费

力扣题目链接

其实这题和最佳时机2相似，**只不过多了个需要交手续费的步骤而已。**

所以在递推公式上，只需要改变以下的步骤：

在不持有股票的状态下：

- 假如卖出股票，那么所得的最大现金为：`dp[i][1]` = `dp[i-1][0]` + `prices[i]` - `fee`);

最大现金为：`dp[i][1]` = Math.max(`dp[i-1][1]` , `dp[i-1][0]` + `prices[i]` - `fee`);

```
// 动归
public int maxProfit(int[] prices, int fee) {
    int len = prices.length;
    int[][] dp = new int[len][2];
    dp[0][0] = -prices[0];
    for (int i = 1; i < len; i++) {
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] - prices[i]);
        dp[i][1] = Math.max(dp[i - 1][0] + prices[i] - fee, dp[i - 1][1]);
    }
    return Math.max(dp[len - 1][0], dp[len - 1][1]);
}
/*
时间复杂度：O(n)
空间复杂度：O(n)
*/
```

总结：

抓住以下问题来解决该专题。

`dp[i][j]`数组的含义；

利用**二维数组**解决该类问题：一个表示持有，一个表示不持有，且 **持有 != 当天购买**，**购买股票 ==》持有**；

将每天的每个股票状态进行分析；

递推公式；