股票问题系列通解（转载翻译）

Storm L4 发布于 2020-08-22 9.1k 阅读

说明

原文出处：Most consistent ways of dealing with the series of stock problems

这篇文章大致是对原文的翻译。和原文相比，这篇文章多了未优化空间的代码，且代码都重新写了，另外更改了部分文字描述。

前言

股票问题一共有六道题，链接如下：

- 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

每个问题都有优质的题解，但是大多数题解没有建立起这些问题之间的联系，也没有给出股票问题系列的通解。这篇文章给出适用于全部股票问题的通解，以及对于每个特定问题的特解。

一、通用情况

这个想法基于如下问题：**给定一个表示每天股票价格的数组，什么因素决定了可以获得的最大收益?**

相信大多数人可以很快给出答案，例如「在哪些天进行交易以及允许多少次交易」。这些因素当然重要，在问题描述中也有这些因素。然而还有一个隐藏但是关键的因素决定了最大收益，下文将阐述这一点。

首先介绍一些符号：

- 用 `n` 表示股票价格数组的长度；
- 用 `i` 表示第 `i` 天（`i` 的取值范围是 `0` 到 `n - 1`）；
- 用 `k` 表示允许的最大交易次数；
- 用 `T[i][k]` 表示在第 `i` 天结束时，**最多**进行 `k` 次交易的情况下可以获得的最大收益。

基准情况是显而易见的：`T[-1][k] = T[i][0] = 0`，表示没有进行股票交易时没有收益（注意第一天对应 `i = 0`，因此 `i = -1` 表示没有股票交易）。现在如果可以将 `T[i][k]` 关联到子问题，例如 `T[i - 1][k]`、`T[i][k - 1]`、`T[i - 1][k - 1]` 等子问题，就能得到状态转移方程，并对这个问题求解。如何得到状态转移方程呢?

最直接的办法是看第 `i` 天可能的操作。有多少个选项？答案是三个：**买入**、**卖出**、**休息**。应该选择哪个操作？答案是：并不知道哪个操作是最好的，但是可以通过计算得到选择每个操作可以得到的最大收益。假设没有别的限制条件，则可以尝试每一种操作，并选择可以最大化收益的一种操作。但是，题目中确实有限制条件，规定不能同时进行多次交易，因此如果决定在第 `i` 天**买入**，在买入之前必须持有 `0` 份股票，如果决定在第 `i` 天**卖出**，在卖出之前必须恰好持有 `1` 份股票。持有股票的数量是上文提及到的隐藏因素，该因素影响第 `i` 天可以进行的操作，进而影响最大收益。

因此对 `T[i][k]` 的定义需要分成两项：

- `T[i][k][0]` 表示在第 `i` 天结束时，**最多**进行 `k` 次交易且在进行操作后持有 `0` 份股票的情况下可以获得的最大收益；
- `T[i][k][1]` 表示在第 `i` 天结束时，**最多**进行 `k` 次交易且在进行操作后持有 `1` 份股票的情况下可以获得的最大收益。

使用新的状态表示之后，可以得到基准情况和状态转移方程。

基准情况：

```
T[-1][k][0] = 0, T[-1][k][1] = -Infinity
T[i][0][0] = 0, T[i][0][1] = -Infinity
```

状态转移方程：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i])
T[i][k][1] = max(T[i - 1][k][1], T[i - 1][k - 1][0] - prices[i])
```

基准情况中，`T[-1][k][0] = T[i][0][0] = 0` 的含义和上文相同，`T[-1][k][1] = T[i][0][1] = -Infinity` 的含义是在没有进行股票交易时不允许持有股票。

对于状态转移方程中的 `T[i][k][0]`，第 `i` 天进行的操作只能是**休息**或**卖出**，因为在第 `i` 天结束时持有的股票数量是 `0`。`T[i - 1][k][0]` 是**休息**操作可以得到的最大收益，`T[i - 1][k][1] + prices[i]` 是**卖出**操作可以得到的最大收益。注意到允许的最大交易次数是不变的，因为每次交易包含两次成对的操作，**买入**和**卖出**。只有**买入**操作会改变允许的最大交易次数。

对于状态转移方程中的 `T[i][k][1]`，第 `i` 天进行的操作只能是**休息**或**买入**，因为在第 `i` 天结束时持有的股票数量是 `1`。`T[i - 1][k][1]` 是**休息**操作可以得到的最大收益，`T[i - 1][k - 1][0] - prices[i]` 是**买入**操作可以得到的最大收益。注意到允许的最大交易次数减少了一次，因为每次**买入**操作会使用一次交易。

为了得到最后一天结束时的最大收益，可以遍历股票价格数组，根据状态转移方程计算 `T[i][k][0]` 和 `T[i][k][1]` 的值。最终答案是 `T[n - 1][k][0]`，因为结束时持有 `0` 份股票的收益一定大于持有 `1` 份股票的收益。

二、应用于特殊情况

上述六个股票问题是根据 `k` 的值进行分类的，其中 `k` 是允许的最大交易次数。最后两个问题有附加限制，包括「冷冻期」和「手续费」。通解可以应用于每个股票问题。

情况一：k = 1

情况一对应的题目是「121. 买卖股票的最佳时机」。

对于情况一，每天有两个未知变量：`T[i][1][0]` 和 `T[i][1][1]`，状态转移方程如下：

```
T[i][1][0] = max(T[i - 1][1][0], T[i - 1][1][1] + prices[i])
T[i][1][1] = max(T[i - 1][1][1], T[i - 1][0][0] - prices[i]) = max(T[i - 1][1][1], -prices[i])
```

第二个状态转移方程利用了 `T[i][0][0] = 0`。

根据上述状态转移方程，可以写出时间复杂度为 $O(n)$ 和空间复杂度为 $O(n)$ 的解法。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        int[][] dp = new int[length][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < 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]);
        }
        return dp[length - 1][0];
    }
}
```

如果注意到第 `i` 天的最大收益只和第 `i - 1` 天的最大收益相关，空间复杂度可以降到 $O(1)$。

Java

```
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int profit0 = 0, profit1 = -prices[0];
        int length = prices.length;
        for (int i = 1; i < length; i++) {
            profit0 = Math.max(profit0, profit1 + prices[i]);
            profit1 = Math.max(profit1, -prices[i]);
        }
        return profit0;
    }
}
```

现在对上述解法进行分析。对于循环中的部分，`profit1` 实际上只是表示到第 `i` 天的股票价格的相反数中的最大值，或者等价地表示到第 `i` 天的股票价格的最小值。对于 `profit0`，只需要决定**卖出**和**休息**中的哪项操作可以得到更高的收益。如果进行**卖出**操作，则**买入**股票的价格为 `profit1`，即第 `i` 天之前（不含第 `i` 天）的最低股票价格。这正是现实中为了获得最大收益会做的事情。但是这种做法不是唯一适用于这种情况的解决方案。读者可能在 这里 找到别的好的解决方案。

情况二：k 为正无穷

情况二对应的题目是「122. 买卖股票的最佳时机 II」。

如果 `k` 为正无穷，则 `k` 和 `k - 1` 可以看成是相同的，因此有 `T[i - 1][k - 1][0] = T[i - 1][k][0]` 和 `T[i - 1][k - 1][1] = T[i - 1][k][1]`。每天仍有两个未知变量：`T[i][k][0]` 和 `T[i][k][1]`，其中 `k` 为正无穷，状态转移方程如下：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i])
T[i][k][1] = max(T[i - 1][k][1], T[i - 1][k - 1][0] - prices[i]) = max(T[i - 1][k][1], T[i - 1][k][0]
```

第二个状态转移方程利用了 `T[i - 1][k - 1][0] = T[i - 1][k][0]`。

根据上述状态转移方程，可以写出时间复杂度为 $O(n)$ 和空间复杂度为 $O(n)$ 的解法。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        int[][] dp = new int[length][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < 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], dp[i - 1][0] - prices[i]);
        }
        return dp[length - 1][0];
    }
}
```

如果注意到第 `i` 天的最大收益只和第 `i - 1` 天的最大收益相关，空间复杂度可以降到 $O(1)$。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int profit0 = 0, profit1 = -prices[0];
        int length = prices.length;
        for (int i = 1; i < length; i++) {
            int newProfit0 = Math.max(profit0, profit1 + prices[i]);
            int newProfit1 = Math.max(profit1, profit0 - prices[i]);
            profit0 = newProfit0;
```

```
            profit1 = newProfit1;
        }
        return profit0;
    }
}
```

这个解法提供了获得最大收益的贪心策略：可能的情况下，在每个局部最小值买入股票，然后在之后遇到的第一个局部最大值卖出股票。这个做法等价于找到股票价格数组中的递增子数组，对于每个递增子数组，在开始位置买入并在结束位置卖出。可以看到，这和累计收益是相同的，只要这样的操作的收益为正。

情况三：k = 2

情况三对应的题目是「123. 买卖股票的最佳时机 III」。

情况三和情况一相似，区别之处是，对于情况三，每天有四个未知变量：`T[i][1][0]`、`T[i][1][1]`、`T[i][2][0]`、`T[i][2][1]`，状态转移方程如下：

```
T[i][2][0] = max(T[i - 1][2][0], T[i - 1][2][1] + prices[i])
T[i][2][1] = max(T[i - 1][2][1], T[i - 1][1][0] - prices[i])
T[i][1][0] = max(T[i - 1][1][0], T[i - 1][1][1] + prices[i])
T[i][1][1] = max(T[i - 1][1][1], T[i - 1][0][0] - prices[i]) = max(T[i - 1][1][1], -prices[i])
```

第四个状态转移方程利用了 `T[i][0][0] = 0`。

根据上述状态转移方程，可以写出时间复杂度为 $O(n)$ 和空间复杂度为 $O(n)$ 的解法。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        int[][][] dp = new int[length][3][2];
        dp[0][1][0] = 0;
        dp[0][1][1] = -prices[0];
        dp[0][2][0] = 0;
        dp[0][2][1] = -prices[0];
        for (int i = 1; i < length; i++) {
            dp[i][2][0] = Math.max(dp[i - 1][2][0], dp[i - 1][2][1] + prices[i]);
            dp[i][2][1] = Math.max(dp[i - 1][2][1], dp[i - 1][1][0] - prices[i]);
            dp[i][1][0] = Math.max(dp[i - 1][1][0], dp[i - 1][1][1] + prices[i]);
            dp[i][1][1] = Math.max(dp[i - 1][1][1], dp[i - 1][0][0] - prices[i]);
        }
        return dp[length - 1][2][0];
    }
}
```

如果注意到第 `i` 天的最大收益只和第 `i - 1` 天的最大收益相关，空间复杂度可以降到 $O(1)$。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int profitOne0 = 0, profitOne1 = -prices[0], profitTwo0 = 0, profitTwo1 = -prices[0];
        int length = prices.length;
        for (int i = 1; i < length; i++) {
            profitTwo0 = Math.max(profitTwo0, profitTwo1 + prices[i]);
            profitTwo1 = Math.max(profitTwo1, profitOne0 - prices[i]);
            profitOne0 = Math.max(profitOne0, profitOne1 + prices[i]);
            profitOne1 = Math.max(profitOne1, -prices[i]);
        }
        return profitTwo0;
    }
}
```

该解法与 这里 给出的解法基本相同。

情况四：k 为任意值

情况四对应的题目是「188. 买卖股票的最佳时机 IV」。

情况四是最通用的情况，对于每一天需要使用不同的 `k` 值更新所有的最大收益，对应持有 `0` 份股票或 `1` 份股票。如果 `k` 超过一个临界值，最大收益就不再取决于允许的最大交易次数，而是取决于股票价格数组的长度，因此可以进行优化。那么这个临界值是什么呢？

一个有收益的交易至少需要两天（在前一天买入，在后一天卖出，前提是买入价格低于卖出价格）。如果股票价格数组的长度为 `n`，则有收益的交易的数量最多为 `n / 2`（整数除法）。因此 `k` 的临界值是 `n / 2`。如果给定的 `k` 不小于临界值，即 `k >= n / 2`，则可以将 `k` 扩展为正无穷，此时问题等价于情况二。

根据状态转移方程，可以写出时间复杂度为 $O(nk)$ 和空间复杂度为 $O(nk)$ 的解法。

Java

```java
class Solution {
    public int maxProfit(int k, int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        if (k >= length / 2) {
            return maxProfit(prices);
        }
        int[][][] dp = new int[length][k + 1][2];
        for (int i = 1; i <= k; i++) {
            dp[0][i][0] = 0;
            dp[0][i][1] = -prices[0];
        }
        for (int i = 1; i < length; i++) {
            for (int j = k; j > 0; j--) {
                dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j][1] + prices[i]);
                dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]);
            }
        }
        return dp[length - 1][k][0];
    }

    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        int[][] dp = new int[length][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
```

如果注意到第 `i` 天的最大收益只和第 `i - 1` 天的最大收益相关，空间复杂度可以降到 $O(k)$。

Java

```java
class Solution {
    public int maxProfit(int k, int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        if (k >= length / 2) {
            return maxProfit(prices);
        }
        int[][] dp = new int[k + 1][2];
        for (int i = 1; i <= k; i++) {
            dp[i][0] = 0;
            dp[i][1] = -prices[0];
        }
        for (int i = 1; i < length; i++) {
            for (int j = k; j > 0; j--) {
                dp[j][0] = Math.max(dp[j][0], dp[j][1] + prices[i]);
                dp[j][1] = Math.max(dp[j][1], dp[j - 1][0] - prices[i]);
            }
        }
        return dp[k][0];
    }
```

```
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int profit0 = 0, profit1 = -prices[0];
        int length = prices.length;
        for (int i = 1; i < length; i++) {
            int newProfit0 = Math.max(profit0, profit1 + prices[i]);
```

如果不根据 `k` 的值进行优化，在 `k` 的值很大的时候会超出时间限制。

该解法与 这里 的解法相似。对交易次数的循环使用反向循环是为了避免使用临时变量。

情况五：k 为正无穷但有冷却时间

情况五对应的题目是「309. 最佳买卖股票时机含冷冻期」。

由于具有相同的 `k` 值，因此情况五和情况二非常相似，不同之处在于情况五有「冷却时间」的限制，因此需要对状态转移方程进行一些修改。

情况二的状态转移方程如下：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i])
T[i][k][1] = max(T[i - 1][k][1], T[i - 1][k][0] - prices[i])
```

但是在有「冷却时间」的情况下，如果在第 `i - 1` 天卖出了股票，就不能在第 `i` 天买入股票。因此，如果要在第 `i` 天买入股票，第二个状态转移方程中就不能使用 `T[i - 1][k][0]`，而应该使用 `T[i - 2][k][0]`。状态转移方程中的别的项保持不变，新的状态转移方程如下：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i])
T[i][k][1] = max(T[i - 1][k][1], T[i - 2][k][0] - prices[i])
```

根据上述状态转移方程，可以写出时间复杂度为 $O(n)$ 和空间复杂度为 $O(n)$ 的解法。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        int[][] dp = new int[length][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < 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], (i >= 2 ? dp[i - 2][0] : 0) - prices[i]);
        }
        return dp[length - 1][0];
    }
}
```

如果注意到第 `i` 天的最大收益只和第 `i - 1` 天和第 `i - 2` 天的最大收益相关，空间复杂度可以降到 $O(1)$。

Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int prevProfit0 = 0, profit0 = 0, profit1 = -prices[0];
        int length = prices.length;
        for (int i = 1; i < length; i++) {
            int nextProfit0 = Math.max(profit0, profit1 + prices[i]);
            int nextProfit1 = Math.max(profit1, prevProfit0 - prices[i]);
            prevProfit0 = profit0;
            profit0 = nextProfit0;
            profit1 = nextProfit1;
        }
```

```
        return profit0;
    }
}
```

dietpepsi 在 这里 分享了一个很好的解法，并加入了思考过程，该解法和上面的解法是相同的。

情况六：k 为正无穷但有手续费

情况六对应的题目是「714. 买卖股票的最佳时机含手续费」。

由于具有相同的 `k` 值，因此情况六和情况二非常相似，不同之处在于情况六有「手续费」，因此需要对状态转移方程进行一些修改。

情况二的状态转移方程如下：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i])
T[i][k][1] = max(T[i - 1][k][1], T[i - 1][k][0] - prices[i])
```

由于需要对每次交易付手续费，因此在每次买入或卖出股票之后的收益需要扣除手续费，新的状态转移方程有两种表示方法。

第一种表示方法，在每次买入股票时扣除手续费：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i])
T[i][k][1] = max(T[i - 1][k][1], T[i - 1][k][0] - prices[i] - fee)
```

第二种表示方法，在每次卖出股票时扣除手续费：

```
T[i][k][0] = max(T[i - 1][k][0], T[i - 1][k][1] + prices[i] - fee)
T[i][k][1] = max(T[i - 1][k][1], T[i - 1][k][0] - prices[i])
```

根据上述状态转移方程，可以写出时间复杂度为 $O(n)$ 和空间复杂度为 $O(n)$ 的解法。

Java | Java

```java
class Solution {
    public int maxProfit(int[] prices, int fee) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int length = prices.length;
        int[][] dp = new int[length][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0] - fee;
        for (int i = 1; i < 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], dp[i - 1][0] - prices[i] - fee);
        }
        return dp[length - 1][0];
    }
}
```

如果注意到第 `i` 天的最大收益只和第 `i - 1` 天的最大收益相关，空间复杂度可以降到 $O(1)$。

Java | Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int profit0 = 0, profit1 = -prices[0] - fee;
        int length = prices.length;
        for (int i = 1; i < length; i++) {
            int newProfit0 = Math.max(profit0, profit1 + prices[i]);
            int newProfit1 = Math.max(profit1, profit0 - prices[i] - fee);
            profit0 = newProfit0;
            profit1 = newProfit1;
        }
        return profit0;
    }
}
```

三、总结

总而言之，股票问题最通用的情况由三个特征决定：当前的天数 `i`、允许的最大交易次数 `k` 以及每天结束时持有的股票数。这篇文章阐述了最大利润的状态转移方程和终止条件，由此可以得到时间复杂度为 $O(nk)$ 和空间复杂度为 $O(k)$ 的解法。该解法可以应用于六个问题，对于最后两个问题，需要将状态转移方程进行一些修改。这里推荐 peterleetcode 的 解法，该解法可以推广到任意的 `k` 值，感兴趣的读者可以进行阅读。

下一篇: 「第 153 场周赛」题解

精选评论(1)

尤达大师 L3 （编辑过）2020-08-27

排名第一的大佬, 膜拜

17 踩 回复

评论(55)

Young 2 天前

你好！请问有人知道为什么情况二，尽可能多的交易就表示无穷次，然后就直接在动态方程中全都不考虑k? ？？

赞 踩 查看 6 条回复 回复

Harrison Wells L1 9 天前

莫非labuladong抄的就是这一篇?

赞 踩 查看 4 条回复 回复

~ L1 2021-02-01

有冷却期的问题（卖出股票后，不能在第二天买入），k不就被限制在 (0,n/2) 了吗，按照情况四来看，状态转移方程的第二个维度[k]、[k-1] 就不能压缩了呀

赞 踩 查看 1 条回复 回复

腐烂的橘子 L6 2020-11-08

赞一个

3 踩 回复

想出去看看 2020-12-28

，大佬

1 踩 回复

Vector L1 2020-12-28

情况四：k 为任意值

从第k次开始到1和从1开始到k次有什么区别吗？为什么要从后往前

```
        for (int i = 1; i < length; i++) {
            for (int j = k; j > 0; j--) {
                dp[j][0] = Math.max(dp[j][0], dp[j][1] + prices[i]);
                dp[j][1] = Math.max(dp[j][1], dp[j - 1][0] - prices[i]);
            }
        }
```

赞 踩 查看 3 条回复 回复

常大伟 2020-11-25

不知道这个算不算是不符合无后效性原则，前面的选择都会影响后面

1 踩 查看 2 条回复 回复

J_hou L1 2020-08-23

很清晰，已经过了这些股票题，，

2 踩 回复

WAN L1 2020-09-01

情况三为什么不是对dp[i][2][0]和dp[i][1][0]取最大呢

赞 踩 查看 5 条回复 回复

Jaan L1 2020-12-10

为什么最多允许的交易次数只和买有关，买和卖是一一对应的，要是我想限制卖的次数而不是买的次数，可以吗？该怎么写状态转移方程

赞 踩 查看 1 条回复 回复

Velvet 2020-10-30

如何证明： 因为结束时持有 0 份股票的收益一定大于持有 1 份股票的收益？

赞 踩 查看 2 条回复 回复

Phil L1 2020-08-29

为什么k要从大到小遍历不是从小到大遍历？ Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]); 最多交易k这种情况下 用到了 最多交易k-1次的结果 可是这时候k-1次还没有结果啊

赞 踩 查看 3 条回复 回复

Gabriel-18 L3 8 天前

😓太🐮了

赞 踩 回复

Gabriel-18 L3 8 天前

大佬还会发其他dp的总结吗

赞 踩 回复