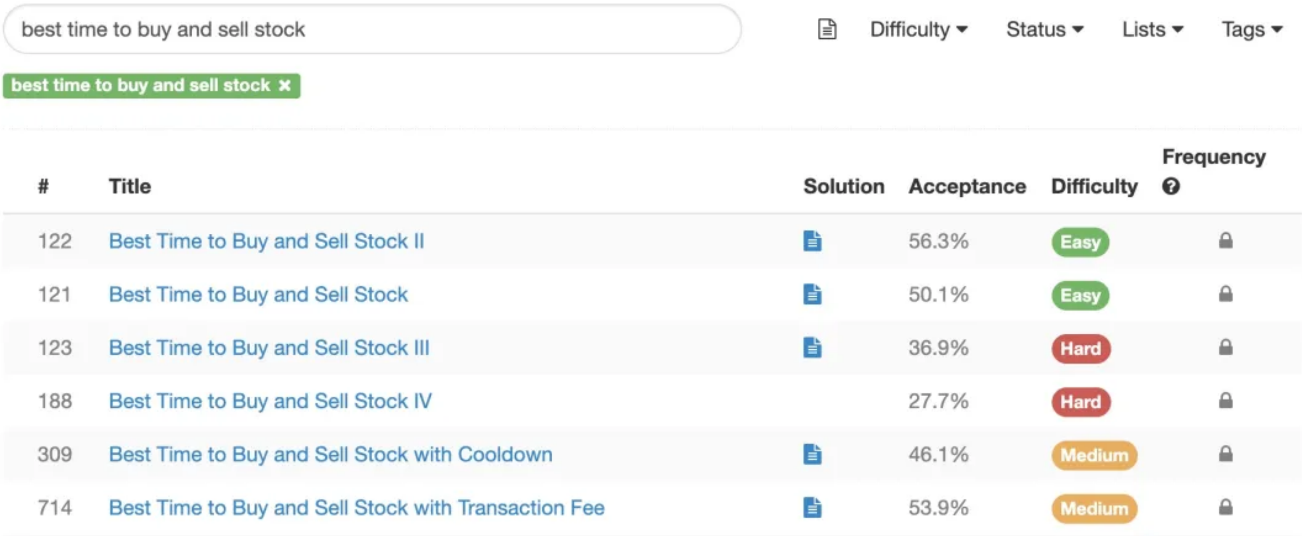
一文教你股票买卖问题实用而装逼的解法

原创 nettee 面向大象编程 6月14日

来自专辑

LeetCode 例题精讲

「股票买卖问题」大概是每个刷 LeetCode 的同学都会遇到的一大拦路虎，特别是其中的第三道题。你是否也曾因为这道题而懵逼呢？



#	Title	Solution	Acceptance	Difficulty	Frequency
122	Best Time to Buy and Sell Stock II		56.3%	Easy	
121	Best Time to Buy and Sell Stock		50.1%	Easy	
123	Best Time to Buy and Sell Stock III		36.9%	Hard	
188	Best Time to Buy and Sell Stock IV		27.7%	Hard	
309	Best Time to Buy and Sell Stock with Cooldown		46.1%	Medium	
714	Best Time to Buy and Sell Stock with Transaction Fee		53.9%	Medium	

股票买卖系列问题

LeetCode 上的股票买卖系列问题一共六道，形成一个巨大的系列，蔚为壮观。系列的前两道题并不难，可以通过思维转换得到解法。然而就在你以为整个系列都可以循序渐进地做出来时，第三道题的难度陡然上升，让人猝不及防。

更令人沮丧的是，这样一道难题，打开讨论区，看到的却是一份异常装逼的题解代码：

```java
public int maxProfit(int[] prices) {
    if (prices.length == 0) return 0;
    int s1 = Integer.MIN_VALUE, s2 = 0,
        s3 = Integer.MIN_VALUE, s4 = 0;
    for (int p : prices) {
        s1 = Math.max(s1, -p);
        s2 = Math.max(s2, s1 + p);
        s3 = Math.max(s3, s2 - p);
        s4 = Math.max(s4, s3 + p);
    }
    return Math.max(0, s4);
}
```

WTF?? 这谜一样的四个变量 `s1` / `s2` / `s3` / `s4` 是怎么回事？这种计算方式怎么能体现题目中「最多买卖两次」的限制？

不要慌张。其实这类问题是有套路的，只要掌握了套路，你也一样能写出这样装逼的解法。这个套路也非常实用，几道股票买卖问题都可以用这个套路解出来。

这样实用而装逼的解法，今天就让我为你细细讲述。本文会介绍股票买卖问题的这个解法中涉及到的几个技巧：

- 动态规划子问题的状态拆解与状态机定义
- DP 数组的特殊值定义
- 动态规划的空间优化技巧

问题解法

我们来求解最典型的股票问题（三），它是其他几道题目解法的基础：

LeetCode 123. Best Time to Buy and Sell Stock III（Hard）

> 给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你最多可以完成**两笔**交易。
>
> **注意：** 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
>
> **示例：**
>
> ```
> 输入: [3,3,5,0,0,3,1,4]
> 输出: 6
> 解释: 在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润
>      随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获
> ```

很多同学可能已经隐约想到这道题是用动态规划来解，但是一直想不出来合适的具体思路。

这道题最大的难点就在于其限制条件「最多完成两笔交易」。如何在动态规划中描述这个限制条件？如何记录股票买卖过程中的「不同状态」？其实，全部的答案就在我们上一篇文章中讨论的技巧：**拆分动态规划的子问题**。

不记得上一篇文章内容的同学可以点这里回顾：

LeetCode 例题精讲 | 17 动态规划如何拆分子问题，简化思路

当然，如果你只想知道股票买卖问题的解法，可以直接往下看。

状态机定义

对于题目中「最多完成两笔交易」这个限制条件，我们可以理解成：**股票买卖的过程，经历了不同的阶段**。

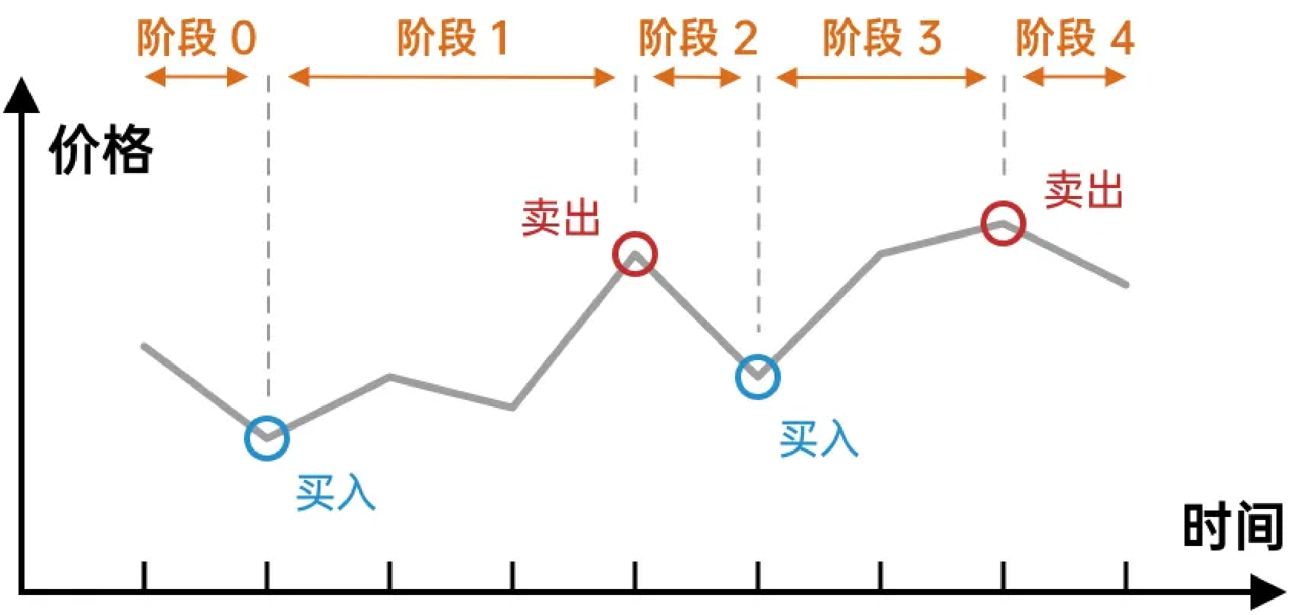
- 在一开始，限制是「最多完成两笔交易」；
- 做完一笔交易之后，限制变成「只做一笔交易」；
- 做完两笔交易之后，限制变成「不能再做交易」。

有的解法选择定义一个参数 k 来表示可以进行交易的数量。不过 k 的取值只有 2、1、0，却要给动态规划增加一个维度，不太划算。我们可以直接定义多个子问题来描述这些不同的阶段。

另外，题目中还有一个条件是「再次购买前必须卖掉之前的股票」，这实际上又给股票买卖过程划分了阶段。我们有「持有股票」和「不持有股票」两种状态。在持有股票的时候，只能卖出，不能买入。不持有股票的时候则反之。

总体来看，做两笔交易，则股票买卖的过程可以分为**五个阶段**：

阶段	可交易次数	持股状态	可买入/卖出
0	2	不持有	可买入
1	1	持有	可卖出
2	1	不持有	可买入
3	0	持有	可卖出
4	0	不持有	可买入



股票买卖过程的五个阶段

对应这五个阶段，我们可以定义五个子问题，分别用 s_0、s_1、s_2、s_3、s_4 来表示。字母 s 代表**状态**，股票买卖阶段的变化，其实就是状态的转移。

例如在阶段 1，我们持有股票，此时如果卖出股票，则变成不持有股票的状态，进入阶段 2。

$s_0 \sim s_4$ 之间的状态转移可以用下面这张图来表示：

子问题间的状态转移关系（状态机）

在每一天，我们既可以选择买入/卖出，又可以选择不进行买卖。选择买入或者卖出的话，就会进入下一个阶段，对应状态转移图中向右的边；选择不买卖的话，会继续待在当前阶段，对应状态转移图中的环路。

这就是所谓的「**状态机 DP**」。定义多个子问题，从另一个角度来看，就是子问题在不同的「状态」间跳来跳去。

在理解了子问题之间的关系之后，我们正式地定义一下子问题和递推关系：子问题 $s_{0/1/2/3/4}(k)$ 分别表示「前 k 天结束后，处于阶段 0/1/2/3/4 时，当前的最大利润」。那么我们有：

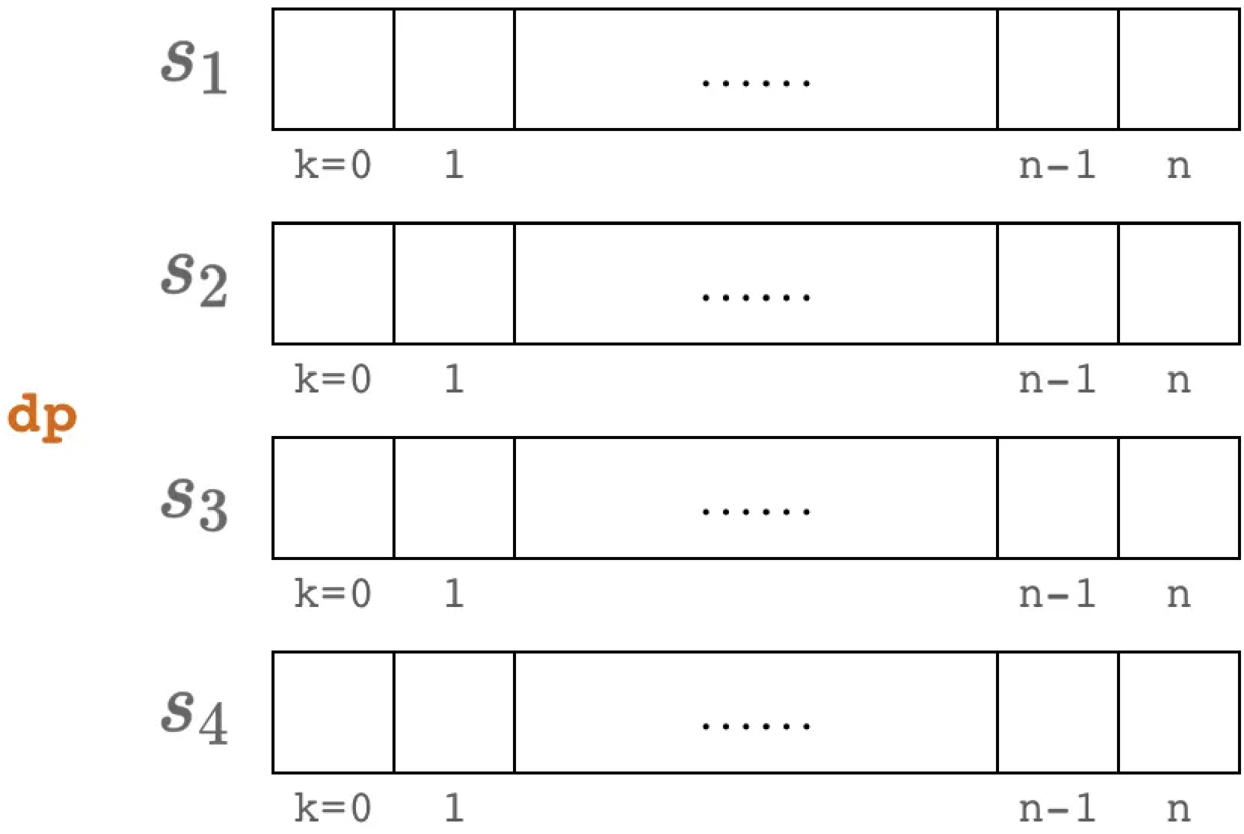
- $s_0(k) = 0$。因为阶段 0 时没有任何买入卖出。
- $s_1(k) = \max\{s_1(k-1), s_0(k-1) - p_k\}$。第 k 天处于阶段 1，可能是前一天处于阶段 1，或者是前一天处于阶段 0，然后买入了第 k 天的股票（利润减去 p_k）。
- $s_2(k) = \max\{s_2(k-1), s_1(k-1) + p_k\}$。第 k 天处于阶段 2，可能是前一天处于阶段 2，或者是前一天处于阶段 1，然后卖出了第 k 天的股票（利润增加 p_k）。
- $s_3(k) = \max\{s_3(k-1), s_2(k-1) - p_k\}$。分析同 $s_1(k)$。
- $s_4(k) = \max\{s_4(k-1), s_3(k-1) + p_k\}$。分析同 $s_2(k)$。

理解 DP 数组

在定义了子问题及其递推关系后，我们还需要搞清楚 DP 数组的计算顺序。

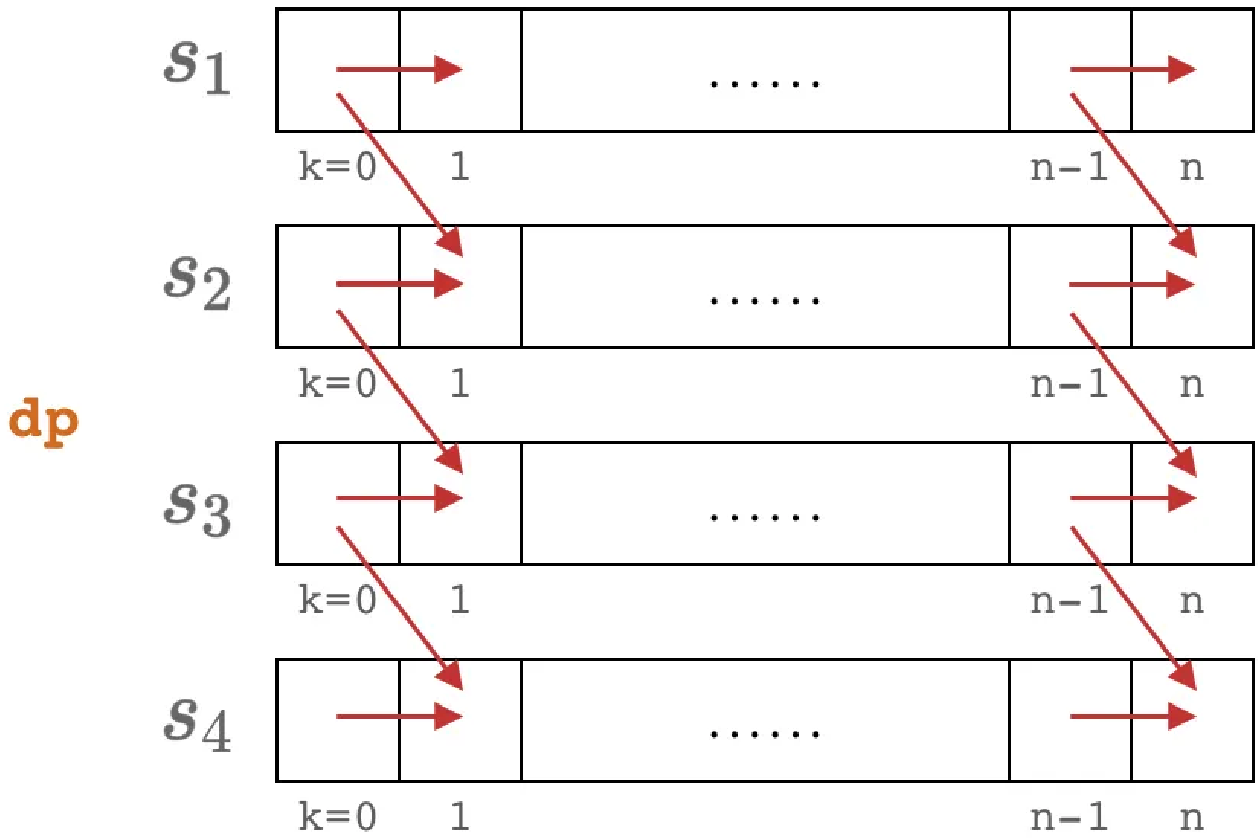
首先，由于 s_0 始终为 0，我们其实不需要计算，直接作为常数 0 代入即可。这样就只剩 $s_1/s_2/s_3/s_4$ 四个子问题了。

四个子问题中，k 的取值范围都是 $0 \le k \le n$。这样我们的 DP 数组是四个长度为 $n+1$ 的一维数组，如下图所示。



DP 数组

接着是 DP 数组中的依赖关系：



DP 数组的依赖关系

可以看出，DP 数组的计算顺序是从左到右、从上到下。我们可以根据这个写出初步的题解代码：

```
// 注意：这段代码并不完整
public int maxProfit(int[] prices) {
    if (prices.length == 0) {
        return 0;
    }

    int n = prices.length;
    int[] s1 = new int[n+1];
    int[] s2 = new int[n+1];
    int[] s3 = new int[n+1];
    int[] s4 = new int[n+1];
    // 注意：这里还缺少 base case 的赋值
    for (int k = 1; k <= n; k++) {
        s1[k] = Math.max(s1[k-1], -p[k-1]);
        s2[k] = Math.max(s2[k-1], s1[k-1] + p[k-1]);
        s3[k] = Math.max(s3[k-1], s2[k-1] - p[k-1]);
        s4[k] = Math.max(s4[k-1], s3[k-1] + p[k-1]);
    }
```

```
        return Math.max(0, Math.max(s2[n], s4[n]));
    }
```

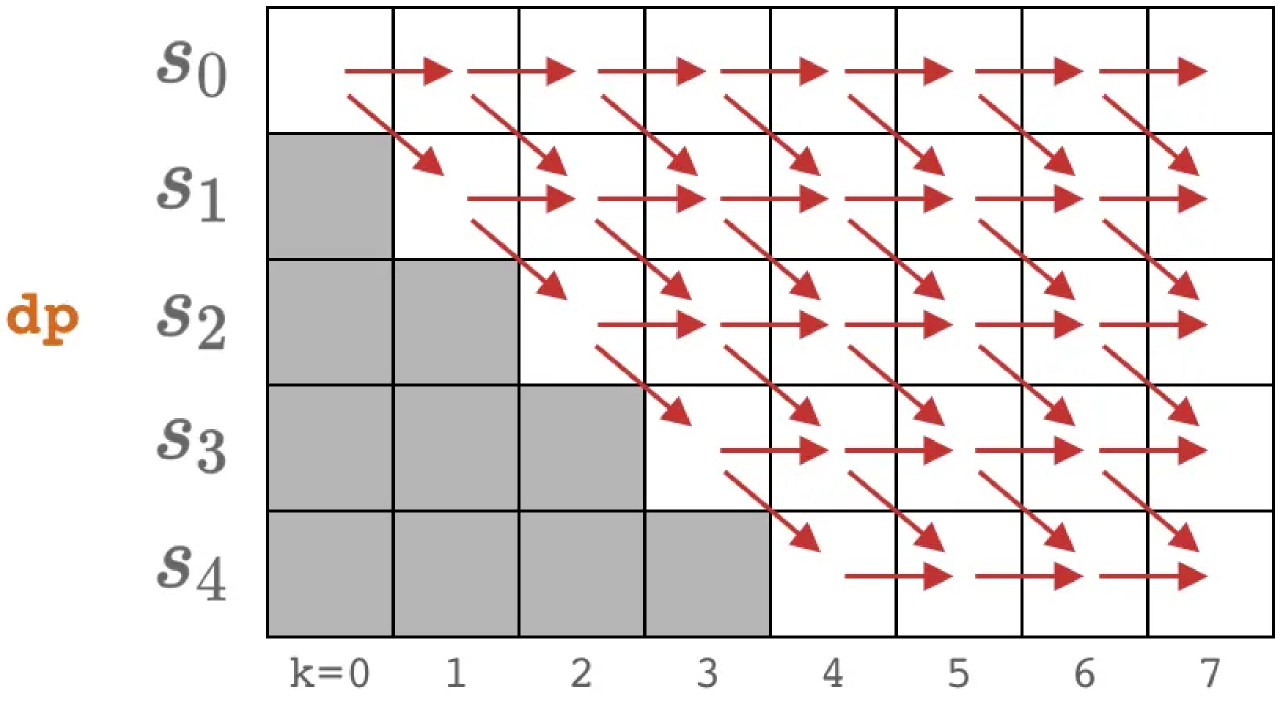
处理 base case

上面的代码还不是很完整，我们还需要填上子问题的 base case 的取值。

对于这道题来说，确定 base case 的取值并不容易。难点在于 DP 数组中的部分元素是无效的。

以 s_2 为例，$s_2(0)$ 的含义应该是：在第 0 天（即一开始），经过一次买入、一次卖出后，所获的最大利润。然而我们在第 0 天显然还不能进行任何买卖，那么 $s_2(0)$ 就是无效元素。我们可以推出，s_2 在 $k \geq 2$ 时才有效。

同样的道理，我们可以计算出 s_1、s_2、s_3、s_4 的 base case 分别是 $s_1(1)$、$s_2(2)$、$s_3(3)$、$s_4(4)$，如下图所示（这里放上 s_0 以方便理解）。



DP 数组中的无效元素

如果用条件判断来处理这些无效元素，代码会变得非常复杂。有没有什么更好的方法呢？

答案是给 $s_1(0)$、$s_2(0)$、$s_3(0)$、$s_4(0)$ 赋特殊值。虽然这些值没有什么实际意义，但不会影响后面有效值的计算，也不会影响最终结果。

既然最终结果要求的是最大利润（max），我们可以给这些无效元素赋一个**比任何可能结果都小的值**：

- 对于 s_2、s_4，这个值是 0。因为这两个状态不持有股票，有效值显然不会低于 0（可以不买也不卖，利润就是 0）。
- 对于 s_1、s_3，这个值是 $-\infty$。因为这两个状态要持有股票，买入后会出现暂时的负利润。

加入这些 base case 之后，我们得到完整的代码：

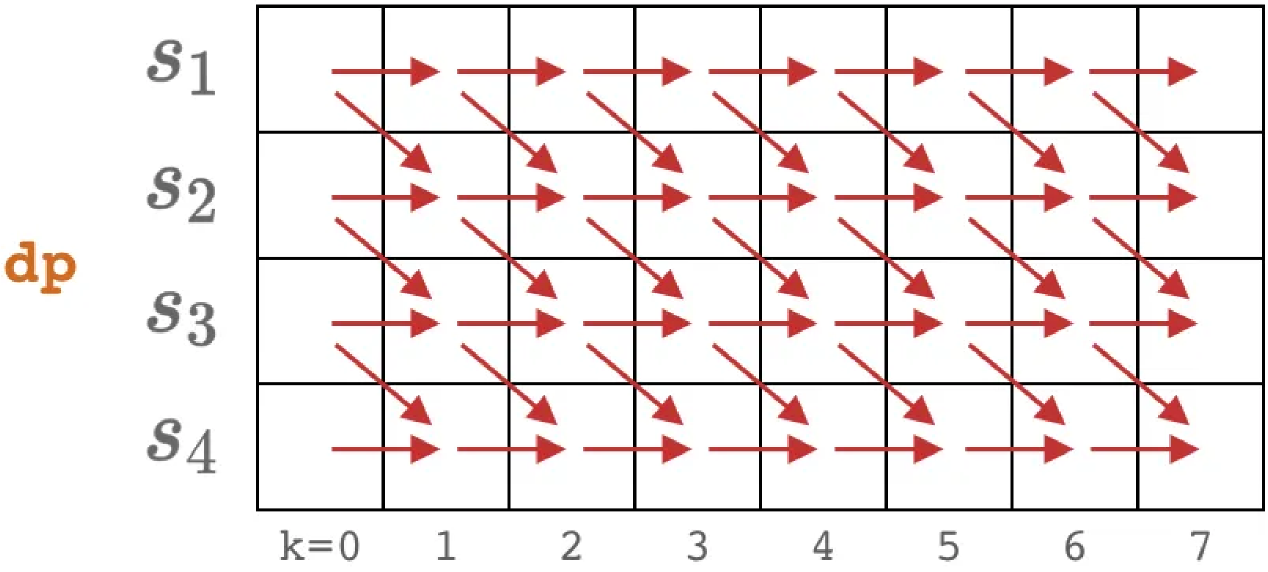
```
public int maxProfit(int[] prices) {
    if (prices.length == 0) {
        return 0;
    }

    int n = prices.length;
    int[] s1 = new int[n+1];
    int[] s2 = new int[n+1];
    int[] s3 = new int[n+1];
    int[] s4 = new int[n+1];
    s1[0] = Integer.MIN_VALUE;
    s2[0] = 0;
    s3[0] = Integer.MIN_VALUE;
    s4[0] = 0;
    for (int k = 1; k <= n; k++) {
        s1[k] = Math.max(s1[k-1], -prices[k-1]);
        s2[k] = Math.max(s2[k-1], s1[k-1] + prices[k-1]);
        s3[k] = Math.max(s3[k-1], s2[k-1] - prices[k-1]);
        s4[k] = Math.max(s4[k-1], s3[k-1] + prices[k-1]);
    }
    return Math.max(0, Math.max(s2[n], s4[n]));
}
```

空间优化

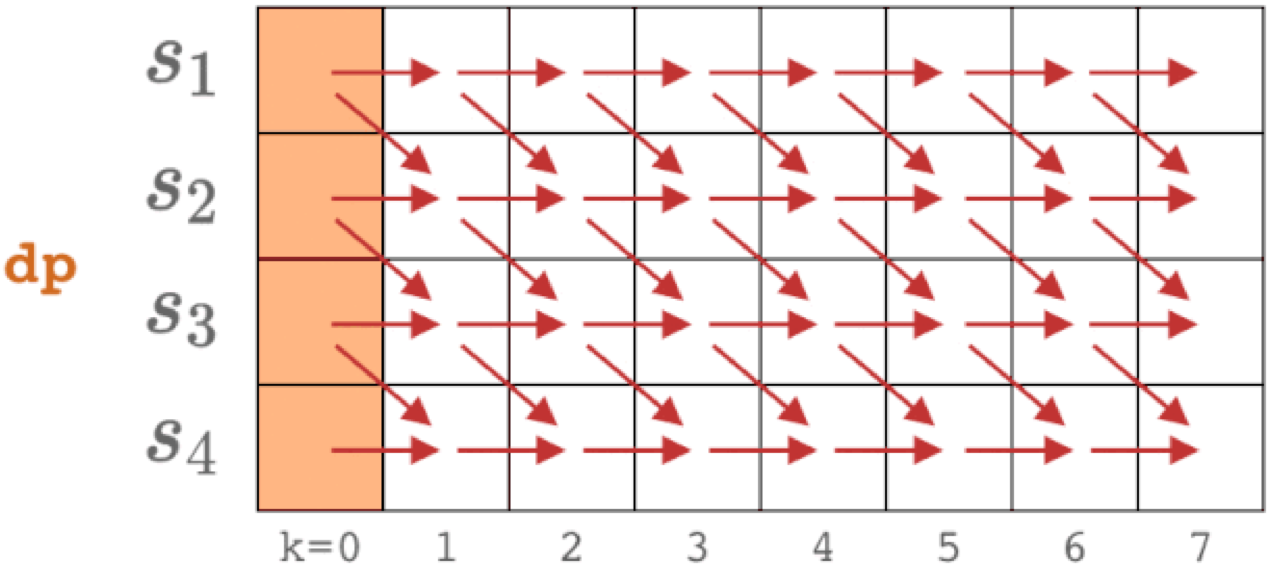
上面的代码已经比较简洁了，不过它和我们一开始展示的装逼型代码还有一点差距。接下来，我们使用一点空间优化的技巧，让代码更加简洁。

回顾一下上面的 DP 数组依赖图：



DP 数组的依赖关系

我们发现，每一列的值都只依赖于上一列的值。这样，我们只需要保存当前一列的值，然后在每一轮迭代中计算下一列的值。



空间优化方案，迭代计算每一列

这样， `s1` 、 `s2` 、 `s3` 、 `s4` 就从一维数组变成了单个变量。

```
int s1 = Integer.MIN_VALUE;
int s2 = 0;
int s3 = Integer.MIN_VALUE;
int s4 = 0;
for (int k = 1; k <= n; k++) {
    s4 = Math.max(s4, s3 + prices[k-1]);
    s3 = Math.max(s3, s2 - prices[k-1]);
```

```
        s2 = Math.max(s2, s1 + prices[k-1]);
        s1 = Math.max(s1, -prices[k-1]);
    }
```

上面的代码中大量出现 `prices[k-1]` 。我们把 `k-1` 替换成 `k` ：

```
    int s1 = Integer.MIN_VALUE;
    int s2 = 0;
    int s3 = Integer.MIN_VALUE;
    int s4 = 0;
    for (int k = 0; k < n; k++) {
        s4 = Math.max(s4, s3 + prices[k]);
        s3 = Math.max(s3, s2 - prices[k]);
        s2 = Math.max(s2, s1 + prices[k]);
        s1 = Math.max(s1, -prices[k]);
    }
```

然后把 for 循环改成 for-each 循环：

```
    int s1 = Integer.MIN_VALUE;
    int s2 = 0;
    int s3 = Integer.MIN_VALUE;
    int s4 = 0;
    for (int p : prices) {
        s4 = Math.max(s4, s3 + p);
        s3 = Math.max(s3, s2 - p);
        s2 = Math.max(s2, s1 + p);
        s1 = Math.max(s1, -p);
    }
```

这样，我们就得到了最终的简化版代码：

```
    public int maxProfit(int[] prices) {
        if (prices.length == 0) {
            return 0;
        }

        int s1 = Integer.MIN_VALUE;
        int s2 = 0;
        int s3 = Integer.MIN_VALUE;
        int s4 = 0;
```

```
        for (int p : prices) {
            s4 = Math.max(s4, s3 + p);
            s3 = Math.max(s3, s2 - p);
            s2 = Math.max(s2, s1 + p);
            s1 = Math.max(s1, -p);
        }
        return Math.max(0, Math.max(s2, s4));
    }
```

这样一步步看下来，是不是感觉开头那个装逼的代码也没有那么难懂了？

总结

本文一步步剖析了股票买卖问题的解题技巧。如果你直接看最终的代码，会觉得「装逼」而放弃这道题。但跟着本文的思路一步步走，会发现这样的代码其实是经过一步步的简化，逐渐变成这个样子的。

LeetCode 讨论区的很多答案喜欢炫耀代码的简洁，比拼行数。但是追求代码的极简并不利于我们掌握问题思路。对于本题而言，其实最值得掌握的是没有经过空间优化的、定义四个一维数组的代码。特别是在面试中，如果你一上来就写出了经过空间优化后的极简代码，面试官可能觉得你是在「背题」，反而对你印象不好。

本文讲述的股票问题的解法有人称之为「状态机 DP」。解法的关键就在于定义多个子问题，然后描述子问题之间的状态转移关系。读完本文的同学，强烈建议把股票买卖问题跟上一篇文章中的例题联系起来看，会让你对这一类问题有更深的理解。

股票买卖系列的其他问题同样可以用这个解题技巧做出来。后面的文章我会给大家展示如何把「最多完成两笔交易」扩充到「最多完成 k 笔交易」的通用版题目，以及带有交易手续费和冷却期的变种题目的求解方法，敬请期待。

往期文章

- **LeetCode 例题精讲 | 17 动态规划如何拆分子问题，简化思路**
- **LeetCode 例题精讲 | 16 最大子数组和：子数组类问题的动态规划技巧**
- **LeetCode 例题精讲 | 14 打家劫舍问题：动态规划的解题四步骤**

我是 nettee，致力于分享面试算法的解题套路，让你真正掌握解题技巧，做到举一反三。我的《LeetCode 例题精讲》系列文章正在写作中，关注我的公众号，获取最新文章。