股票问题总结

状态机, DP table

一、穷举框架

利用「状态」进行穷举。我们具体到每一天，看看总共有几种可能的「状态」，再找出每个「状态」对应的「选择」。我们要穷举所有「状态」，穷举的目的是根据对应的「选择」更新状态。听起来抽象，你只要记住「状态」和「选择」两个词就行，下面实操一下就很容易明白了。

```
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][...] = 择优(选择1，选择2...)
```

比如说这个问题，每天都有三种「选择」：买入、卖出、无操作，我们用 buy, sell, rest 表示这三种选择。但问题是，并不是每天都可以任意选择这三种选择的，因为 sell 必须在 buy 之后，buy 必须在 sell 之后。那么 rest 操作还应该分两种状态，一种是 buy 之后的 rest（持有了股票），一种是 sell 之后的 rest（没有持有股票）。而且别忘了，我们还有交易次数 k 的限制，就是说你 buy 还只能在 k > 0 的前提下操作。

很复杂对吧，不要怕，我们现在的目的只是穷举，你有再多的状态，老夫要做的就是一把梭全部列举出来。这个问题的「状态」有三个，第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（即之前说的 rest 的状态，我们不妨用 1 表示持有，0 表示没有持有）。然后我们用一个三维数组就可以装下这几种状态的全部组合(**状态定义**):

```
dp[i][k][0 or 1]
0 <= i <= n-1, 1 <= k <= K
n 为天数，大 K 为最多交易数
此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:
 for 1 <= k <= K:
 for s in {0, 1}:
 dp[i][k][s] = max(buy, sell, rest)
```

而且我们可以用自然语言描述出每一个状态的含义，比如说 **dp[3][2][1]** 的含义就是：**今天是第三天，我现在手上持有着股票，至今最多进行 2 次交易，而其值为该状态下的最优解。**

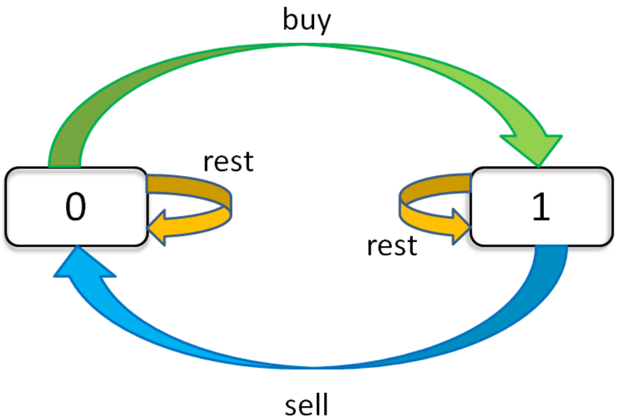
再比如 **dp[2][3][0]** 的含义：**今天是第二天，我现在手上没有持有股票，至今最多进行 3 次交易，而其值为该状态下的最优解。**很容易理解，对吧？

我们想求的最终答案是 dp[n - 1][K][0]，即最后一天，最多允许 K 次交易，最多获得多少利润。读者可能问为什么不是 dp[n - 1][K][1]？因为 [1] 代表手上还持有股票，[0] 表示手上的股票已经卖出去了，很显然后者得到的利润一定大于前者。

记住如何解释「状态」，一旦你觉得哪里不好理解，把它翻译成自然语言就容易理解了。

二、状态转移框架

现在，我们完成了「状态」的穷举，我们开始思考每种「状态」有哪些「选择」，应该如何更新「状态」。只看「持有状态」，可以画个状态转移图。



通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来写一下**状态转移方程**：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
              max( 选择 rest , 选择 sell )

解释：今天我没有持有股票，有两种可能：
要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；
要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
              max( 选择 rest , 选择 buy )

解释：今天我持有着股票，有两种可能：
要么我昨天就持有着股票，然后今天选择 rest，所以我今天还持有着股票；
要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。
```

这个解释应该很清楚了，如果 buy，就要从利润中减去 prices[i]，如果 sell，就要给利润增加 prices[i]。今天的最大利润就是这两种可能选择中较大的那个。而且注意 k 的限制，我们在选择 buy 的时候，把 k 减小了 1，很好理解吧，当然你也可以在 sell 的时候减 1，一样的。

现在，我们已经完成了动态规划中最困难的一步：状态转移方程。如果之前的内容你都可以理解，那么你已经可以秒杀所有问题了，只要套这个框架就行了。不过还差最后一点点，就是定义 base case，即最简单的情况(**边界条件**)。

```
dp[-1][k][0] = 0
解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0 。
dp[-1][k][1] = -infinity
解释：还没开始的时候，是不可能持有股票的，用负无穷表示这种不可能。
dp[i][0][0] = 0
解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0 。
dp[i][0][1] = -infinity
解释：不允许交易的情况下，是不可能持有股票的，用负无穷表示这种不可能。
```

把上面的状态转移方程总结一下：

```
base case:
dp[-1][k][0] = dp[i][0][0] = 0
dp[-1][k][1] = dp[i][0][1] = -infinity

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])
```

三、题解

- [ ] **买卖股票的最佳时机** **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][0][0] - prices[i])
 = max(dp[i-1][1][1], -prices[i])
解释：k = 0 的 base case，所以 dp[i-1][0][0] = 0。

```

```
现在发现 k 都是 1，不会改变，即 k 对状态转移已经没有影响了。
可以进行进一步化简去掉所有 k：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], -prices[i])
```

直接写出代码：

```
int n = prices.length();
int dp[n][2];
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], -prices[i]);
}
return dp[n - 1][0];
```

显然 i=0 时 dp[i-1] 是不合法的。这是因为我们没有对 i 的 base case 进行处理。可以这样处理：

```
int maxProfit(vector<int>& prices) {
    int n=prices.size();
    if(n==0) return 0;
    int dp[n][2];
    /** base case:
 * 已知： dp[-1][0]=0; dp[-1][1]=INT_MIN(-∞).
     * 可推出：
 * dp[0][0]=max(dp[-1][0], dp[-1][1]+prices[0])
   *          =max(0, -∞+prices[0])
    *          =0.
  * dp[0][1]=max(dp[-1][1], dp[-1][0]-prices[0])
  * =max(-∞, 0-prices[0])
  *          =-prices[0].
  */

     dp[0][0]=0;dp[0][1]=-prices[0];
     for(int i=1;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], -prices[i]);
     }
     return dp[n-1][0];
 }
```

第一题就解决了，但是这样处理 base case 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，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间复杂度降到 O(1):

```
int maxProfit(vector<int>& prices) {
    int n=prices.size();
    if(n==0) return 0;
    int dp_i_0=0, dp_i_1=INT_MIN;

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

买卖股票的最佳时机 II k=∞

如果 k 为正无穷，那么就可以认为 k 和 k - 1 是一样的。可以这样改写框架:

```
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])

我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态了：
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])
```

直接翻译成代码:

```
int maxProfit(vector<int>& prices) {
    int n=prices.size();
    if(n==0) return 0;
    int dp[n][2];
    dp[0][0]=0;dp[0][1]=-prices[0];
    for(int i=1;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]);
    }
```

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

空间复杂度O(1)，可以发现与第一题相比，差别仅在dp[i-1][k-1][0]上！

```
int maxProfit(vector<int>& prices) {
    int dp_i_0=0,dp_i_1=INT_MIN;
    for(int i=0;i<prices.size();i++){
        int temp=dp_i_0;
        dp_i_0=max(dp_i_0, dp_i_1+prices[i]);
        dp_i_1=max(dp_i_1, temp-prices[i]);
    }
    return dp_i_0;
}
```

- [ ] **最佳买卖股票时机含冷冻期** k=∞

每次 sell 之后要等一天才能继续交易。只要把这个特点融入上一题的状态转移方程即可：

```
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])
解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1 。
```

转换为代码：

```
int maxProfit(vector<int>& prices) {
    int n=prices.size();
    if(n==0||n==1) return 0;
    int dp[n][2];
    /** base case:
      * 已知：dp[0][0]=0;dp[0][1]=-prices[0];
      * 可推出：
      * dp[1][0]=max(dp[0][0], dp[0][1]+prices[1])
      *          =max(0, -prices[0]+prices[1])
       * dp[1][1]=max(dp[0][1], dp[0][0]-prices[1])
       *          =max(-prices[0], 0-prices[1])
       */
     dp[0][0]=0;dp[0][1]=-prices[0];
     dp[1][0]=max(0, -prices[0]+prices[1]);
```

```
    dp[1][1]=max(-prices[0], 0-prices[1]);
    for(int i=2;i<n;i++){
        dp[i][0]=max(dp[i-1][0], dp[i-1][1]+prices[i]);//保持或卖出
        dp[i][1]=max(dp[i-1][1], dp[i-2][0]-prices[i]);//保持或买入
    }
    return dp[n-1][0];
}
```

空间复杂度优化O(1)：

```
int maxProfit(vector<int>& prices) {
    //dp_pre_0即dp[i-2][0]，初值dp[-2][0]=0
 int dp_i_0=0, dp_i_1=INT_MIN, dp_pre_0=0;

    for(int i=0;i<prices.size();i++){
        int temp=dp_i_0;
        dp_i_0=max(dp_i_0, dp_i_1+prices[i]);//保持或卖出
        dp_i_1=max(dp_i_1, dp_pre_0-prices[i]);//保持或买入
        dp_pre_0=temp;
     }
     return dp_i_0;
}
```

- [ ] **买卖股票的最佳时机含手续费 k=∞**

每次交易要支付手续费，只要把手续费从利润中减去即可。改写方程：

```
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)
解释：相当于买入股票的价格升高了。
在第一个式子里减也是一样的，相当于卖出股票的价格减小了。
```

在题II的基础上稍作修改，即可得到代码：

```
int maxProfit(vector<int>& prices, int fee) {
    int n =prices.size();
    if(n==0) return 0;
    int dp[n][2];
    dp[0][0]=0;dp[0][1]=-prices[0];
```

```
for(int i=1;i<n;i++){
    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]);
}
 return dp[n-1][0];
}
```

空间复杂度 O(1):

```
略
```

买卖股票的最佳时机 III k=2

k = 2 和前面题目的情况稍微不同，因为上面的情况都和 k 的关系不太大。要么 k 是正无穷，状态转移和 k 没关系了；要么 k = 1，跟 k = 0 这个 base case 挨得近，最后也没有存在感。

这道题 k = 2 和后面要讲的 k 是任意正整数的情况中，对 k 的处理就凸显出来了。我们直接写代码，边写边分析原因。

```
原始的动态转移方程，没有可化简的地方
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])
```

还记得前面总结的「穷举框架」吗？就是说我们必须穷举所有状态。其实我们之前的解法，都在穷举所有状态，只是之前的题目中 k 都被化简掉了。这道题由于没有消掉 k 的影响，所以必须要对 k 进行穷举：

```
int maxProfit(vector<int>& prices) {
    int n=prices.size(), max_k=2;
    if(n==0) return 0;
    int dp[n][max_k+1][2];
    /**  base case:
      *  已知:
      *  dp[-1][k][0]=dp[i][0][0]=0;
      *  dp[-1][k][1]=dp[i][0][1]=INT_MIN(-∞);
      *  可推出:
       *  dp[0][2][0]=max(dp[-1][2][0], dp[-1][2][1]+prices[0])
```

```
         *              =max(0, -∞+prices[0])
         *              =0;
         *  dp[0][1][0]=max(dp[-1][1][0], dp[-1][1][1]+prices[0])
         *              =max(0, -∞+prices[0])
         *              =0;
         *  dp[0][2][1]=max(dp[-1][2][1], dp[-1][2-1][0]-prices[0])
         *              =max(-∞, 0-prices[0])
         *              =-prices[0];
         *  dp[0][1][1]=max(dp[-1][1][1], dp[-1][1-1][0]-prices[0]);
         *              =max(-∞, 0-prices[0])
         *              =-prices[0];
         */
        dp[0][0][0]=dp[0][1][0]=dp[0][2][0]=0;
        //dp[0][0][1]=INT_MIN; 其实用不着
        dp[0][1][1]=dp[0][2][1]=-prices[0];
        for(int i=1;i<n;i++){
            dp[i][0][0]=0;//dp[i][0][1]=INT_MIN; 其实用不着
            for(int k=max_k;k>0;k--){
                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]);//交易次数-1
            }
        }
        return dp[n-1][max_k][0];
}
```

这里 k 取值范围比较小，所以可以不用 for 循环，直接把 k = 1 和 2 的情况手动列举出来也可以：

```
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])
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])
 = max(dp[i-1][1][1], 0 - prices[i])
注意：这里的dp[i][0][0]=0

int maxProfit(vector<int>& prices) {
     int dp_i_1_0=0, dp_i_2_0=0;
      int dp_i_1_1=INT_MIN, dp_i_2_1=INT_MIN;
      for(int i=0;i<prices.size();i++){
          dp_i_2_0=max(dp_i_2_0, dp_i_2_1+prices[i]);
```

```
        dp_i_2_1=max(dp_i_2_1, dp_i_1_0-prices[i]);
        dp_i_1_0=max(dp_i_1_0, dp_i_1_1+prices[i]);
        dp_i_1_1=max(dp_i_1_1, -prices[i]); //dp_i_0_0=0
    }
    return dp_i_2_0;
}
```

有状态转移方程和含义明确的变量名指导，相信你很容易看懂。其实我们可以故弄玄虚，把上述四个变量换成 a, b, c, d。这样当别人看到你的代码时就会一头雾水，大惊失色，不得不对你肃然起敬。

卖股票的最佳时机 IV

有了上一题 k = 2 的铺垫，这题应该和上一题的第一个解法没啥区别。但是出现了一个超内存的错误，原来是传入的 k 值会非常大，dp 数组太大了。现在想想，交易次数 k 最多有多大呢?

一次交易由买入和卖出构成，至少需要两天。所以说有效的限制 k 应该不超过 n/2，如果超过，就没有约束作用了，相当于 k = +infinity。这种情况是之前解决过的，可以直接复用代码。

```
int maxProfit_k_any(vector<int>& prices) {
    int dp_i_0=0,dp_i_1=INT_MIN;
    for(int i=0;i<prices.size();i++){
        int temp=dp_i_0;
        dp_i_0=max(dp_i_0, dp_i_1+prices[i]);
        dp_i_1=max(dp_i_1, temp-prices[i]);
    }
    return dp_i_0;
}

int maxProfit(int maxk, vector<int>& prices) {
    int n=prices.size();
    if(n==0) return 0;
    if(maxk>n/2) return maxProfit_k_any(prices);//k=∞
    int dp[n][maxk+1][2];
    //base case:
```

```
    for(int k=0;k<=maxk;k++)
        dp[0][k][0]=0;
    for(int k=1;k<=maxk;k++)
        dp[0][k][1]=-prices[0];
    //dp[0][0][1]=INT_MIN;

    for(int i=1;i<n;i++){
        dp[i][0][0]=0;//dp[i][0][1]=INT_MIN;
        for(int k=maxk;k>0;k--){
            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]);//交易次数-1
        }
    }
    return dp[n-1][maxk][0];
}
```

四、最后总结

本文给大家讲了如何通过状态转移的方法解决复杂的问题，用一个状态转移方程秒杀了 6 道股票买卖问题，现在想想，其实也不算难对吧？这已经属于动态规划问题中较困难的了。

关键就在于列举出所有可能的「状态」，然后想想怎么穷举更新这些「状态」。一般用一个多维 dp 数组储存这些状态，从 base case 开始向后推进，推进到最后的状态，就是我们想要的答案。想想这个过程，你是不是有点理解「动态规划」这个名词的意义了呢？

具体到股票买卖问题，我们发现了三个状态，使用了一个三维数组，无非还是穷举 + 更新，不过我们可以说的高大上一点，这叫「三维 DP」，怕不怕？这个大实话一说，立刻显得你高人一等，名利双收有没有。

所以，大家不要被各种高大上的名词吓到，再多的困难问题，奇技淫巧，也不过是基本套路的不断升级组合产生的。只要把住算法的底层原理，即可举一反三，逐个击破。