题目举例 Leetcode上一共6个股票买卖问题， 这里是第三题的2个case

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 **k** 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 1:

```
输入: [2,4,1], k = 2
输出: 2
解释: 在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。
```

示例 2:

```
输入: [3,2,6,5,0,3], k = 2
输出: 7
解释: 在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 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 = 3-0 = 3 。
```

第一题是只进行一次交易，相当于 k = 1；第二题是不限交易次数，相当于 k = +infinity（正无穷）；**第三题是只进行 2 次交易，相当于 k = 2**；剩下两道也是不限次数，但是加了交易「冷冻期」和「手续费」的额外条件，其实就是第二题的变种，都很容易处理。

穷举框架

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

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：

```
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，至今最多再进行 2 次交易。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，至今最多再进行 3 次交易。

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。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

状态转移框架

```
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。
```

base case

```
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。
```

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])
```

第一题， 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])
```

代码

```
int n =prices.length;
int[][]dp = new int[n][2];
for (int i = 0; i < n; i++){
 if (i - 1 == -1) {
 dp[i][0] = 0;
 // 解释：
 // dp[i][0]
 // = max(dp[-1][0], dp[-1][1] + prices[i])
 // = max(0, -infinity + prices[i]) = 0
  dp[i][1] = -prices[i];
  //解释：
  // dp[i][1]
  // = max(dp[-1][1], dp[-1][0] - prices[i])
  // = max(-infinity, 0 - prices[i])
  // = -prices[i]
  continue;}
  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] //最后一天，手上不持有股票
```

因为新状态只和前一个状态有关，所以这里可以把dp[i][0]和dp[i][1]变为dp_i_0 dp_i_1 ,不断通过循环更新即可

```
// k == 1 空间复杂度 $O(1)$
int maxProfit_k_1(int[] prices) {
 int n = prices.length;
 // base case: dp[-1][0] = 0, dp[-1][1] = -infinity
 int dp_i_0 = 0, 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_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
 // dp[i][1] = max(dp[i-1][1], -prices[i])
  dp_i_1 = Math.max(dp_i_1, -prices[i]);
  }
  return dp_i_0;
 }
```

第二题，k = +infinity

```
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])
```

代码就是

```
int maxProfit_k_inf(int[] prices) {
 int n = prices.length;
 int dp_i_0 = 0, 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]);
 //注意temp，不能直接用dp_i_0，因为dp_i_0 在上一步第六行已经被更新过了
 //得用每一轮更新之前的dp_i_0
```

```
    }
    return dp_i_0;
}
```

第三题，k = +infinity with cooldown

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 。
```

代码

```
int maxProfit_with_cool(int[] prices) {
  int n = prices.length;
  int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
  int dp_pre_0 = 0; // 代表 dp[i-2][0]
  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, dp_pre_0 - prices[i]);
  dp_pre_0 = temp;
   }
   return dp_i_0;
}
```

第四题，k = +infinity with fee

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。
```

```
int maxProfit_with_fee(int[] prices, int fee) {
  int n = prices.length;
  int dp_i_0 = 0, 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;
}
```

第五题，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。

注意，比如用穷举法，因为这里的K是变量，不再是常量了。最多可以买卖2次，就要考虑到买卖1次的情况

```
int max_k = 2;
int[][][] dp = new int[n][max_k + 1][2];
for (int i = 0; i < n; i++) {
  for (int k = max_k; k >= 1; k--) {
  if (i - 1 == -1) { /*处理 base case */ }
```

```
        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]);
    }
}
// 穷举了 n × max_k × 2 个状态，正确。
return dp[n - 1][max_k][0];
```

第六题 k = any integer

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。

```
int maxProfit_k_any(int max_k, int[] prices) {
    int n = prices.length;
    if (max_k > n / 2)
        return maxProfit_k_inf(prices);//第二题的K，无穷大

    int[][][] dp = new int[n][max_k + 1][2];
    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) { /* 处理 base case */ }
            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]);
        }
    return dp[n - 1][max_k][0];
}
```

https://github.com/labuladong/fucking-algorithm/blob/master/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92%E7%B3%BB%E5%88%97/%E5%9B%A2%E7%81%A