8 序列型动规 Leetcode 买卖股票系列

笔记本： DP Note
创建时间： 10/23/2019 10:37 PM **更新时间：** 10/26/2019 2:12 PM
作者： tanziqi1756@outlook.com

---

121. Best Time to Buy and Sell Stock

难度 简单 588 收藏 分享 切换为中文 关注

题目描述 评论 (455) 题解(110) New 提交记录

Say you have an array for which the i^{th} element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
             Not 7-1 = 6, as selling price needs to be larger than buying price.
```

Example 2:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

只能买卖一笔

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

        int valley = prices[0];
        int ans = 0;

        // dp
        // 假设prices[j]为卖出点，不断动态更新j点之前的最小值
        for( int j = 1; j < prices.length; j++ ) {
            // 注意要先更新收益
            ans = Math.max(ans, prices[j] - valley);
            // 再更新最低点
            // 因为要先买后卖
            valley = Math.min(valley, prices[j]);
        }

        return ans;

    }
}

```

升级版，任意多笔transactons.
反而更简单，贪心算法，今天买明天卖。

122. Best Time to Buy and Sell Stock II

难度 简单 496 收藏 分享 切换为中文 关注

题目描述 评论 (499) 题解(97) New 提交记录

Say you have an array for which the i^{th} element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
             Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
```

Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
             Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are
             engaging multiple transactions at the same time. You must sell before buying again.
```

Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

```java
class Solution {
    public int maxProfit(int[] prices) {
        int profit = 0;
        for( int i = 0; i < prices.length - 1; i++ ) {
            if( prices[i + 1] > prices[i] ) {
                profit += prices[i+1] - prices[i];
            }
        }
        return profit;
    }
}
```

只能最多完成两笔交易
这就有难度了，因为有时候一笔交易就能够达到最很大的收益，
但还有一笔收益在一个峰值之后。

123. Best Time to Buy and Sell Stock III

难度 困难 225 收藏 分享 切换为中文 关注

题目描述 评论 (108) 题解(29) New 提交记录

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most *two* transactions.

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3),
profit = 3-0 = 3.
             Then buy on day 7 (price = 1) and sell on day 8 (price = 4),
profit = 4-1 = 3.
```

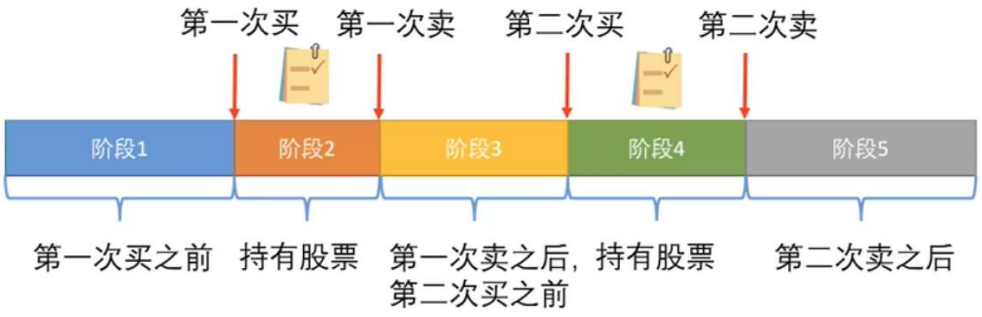
Example 2:

```
Input: [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5),
profit = 5-1 = 4.
             Note that you cannot buy on day 1, buy on day 2 and sell
them later, as you are
             engaging multiple transactions at the same time. You must
sell before buying again.
```

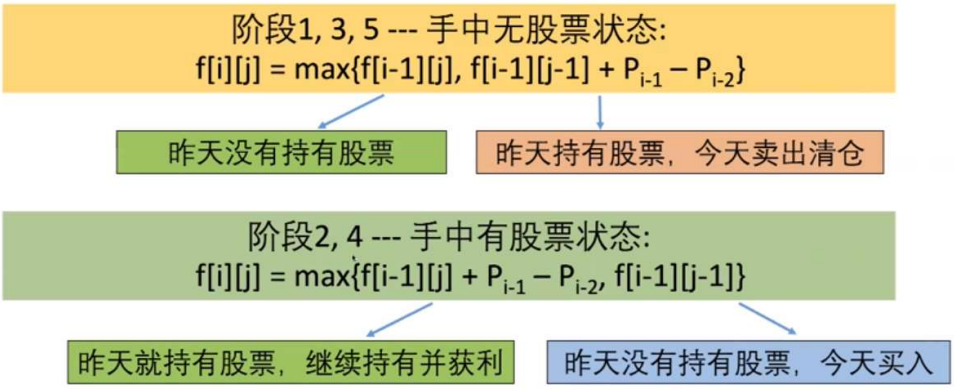
Example 3:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

- 状态：f[i][j]表示前i天(第i-1天)结束后，在阶段j的最大获利



- f[i][j]:前i天(第i-1天)结束后，处在阶段j, 最大获利



刚开始（前0天）处于阶段1

- $f[0][1] = 0$
- $f[0][2] = f[0][3] = f[0][4] = f[0][5] = -\infty$

阶段1, 3, 5: $f[i][j] = \max\{f[i-1][j], f[i-1][j-1] + P_{i-1} – P_{i-2}\}$

阶段2, 4: $f[i][j] = \max\{f[i-1][j] + P_{i-1} – P_{i-2}, f[i-1][j-1]\}$

```java
class Solution {
    public int maxProfit(int[] prices) {
        int N = prices.length;
        if( N == 0 ) {
            return 0;
        }
        // 阶段1：第一次买入前
        // 阶段2：第一次持有股票
        // 阶段3：第一次卖出股票后，第二次买入前
        // 阶段4：第二次持有股票
        // 阶段5：第二次卖出股票
        // dp[i][j] 表示第i-1天结束后，处于阶段j时的最大获利
        int[][] dp = new int[N+1][5+1];
        for( int i = 0; i <= 5; i++ ) {
            dp[0][i] = Integer.MIN_VALUE;
        }
        dp[0][1] = 0;

        for( int i = 1; i <= N; i++ ) {
            // 阶段1，3，5：  dp[i][j] = max{dp[i-1], dp[i-1][j-1] + prices[i-1] - prices[i-2]}
            for( int j = 1; j <= 5; j += 2 ) {
                dp[i][j] = dp[i-1][j];
                // sell
                if( j > 1 && i > 1 && dp[i-1][j-1] != Integer.MIN_VALUE) {
                    dp[i][j] = Math.max(dp[i][j], dp[i-1][j-1] + prices[i-1] - prices[i-2]);
                }
            }
            // 阶段2，4：  dp[i][j] = max{dp[i-1][j] + prices[i-1] - prices[i-2], dp[i-1][j-1]}
            for( int j = 2; j <= 5; j += 2 ) {
                dp[i][j] = dp[i-1][j-1];
                if( i > 1 && dp[i-1][j] != Integer.MIN_VALUE ) {
                    dp[i][j] = Math.max(dp[i][j], dp[i-1][j] + prices[i-1] - prices[i-2]);
                }
            }
        }

        int res = 0;
        for( int j = 1; j <= 5; j += 2 ) {
            res = Math.max(res, dp[N][j]);
        }
        return res;
    }

}
```

/*

```
Author: Ziqi Tan
*/
public class Solution {
     public int maxProfit(int[] prices) {
          System.out.println(Arrays.toString(prices));
        int N = prices.length;
        if( N == 0 ) {
            return 0;
         }
        // 阶段1：第一次买入前
        // 阶段2：第一次持有股票
        // 阶段3：第一次卖出股票后，第二次买入前
        // 阶段4：第二次持有股票
        // 阶段5：第二次卖出股票
        // 状态：
        //dp[i][j] 表示第i-1天结束后，处于阶段j
时的最大获利
        // 转移方程：
        // 某一天的阶段1，等于前一天的阶段1
        // 某一天的阶段2，等于前一天的阶段2，或者
前一天的阶段1转移到今天的阶段2
        // 某一天的阶段3，等于前一天的阶段3，或者
前一天的阶段2转移到今天的阶段3，计算获利
        // 某一天的阶段4，等于前一天的阶段4，或者
前一天的阶段3转移到今天的阶段4
        // 某一天的阶段5，等于前一天的阶段5，或者
前一天的阶段4转移到今天的阶段5，计算获利
        int[][] dp = new int[N+1][5+1];
        // 5+1 是为了方便直接使用index表示阶段
 (数组第一列是没有意义的)
        for( int i = 0; i <=5; i++ ) {
            dp[0][i] = Integer.MIN_VALUE;
            // 由于第0天只能处于阶段1，所以阶段
2345都是无意义的，初始化为0
         }
        dp[0][1] = 0;
         print(dp);
        for( int i = 1; i <= N; i++ ) {
```

```
            // 阶段1, 3, 5: dp[i][j] =
max{dp[i-1],dp[i-1][j-1] + prices[i-1] -
prices[i-2]}
            for(int j = 1;j <= 5;j += 2 ) {
                dp[i][j] =dp[i-1][j];// 先初始
化为前一天的状态
                // sell
                // 总不能第一天买第一天卖，所以i
> 1 && j > 1
                if(j > 1 &&i > 1 &&dp[i-1][j-
1] != Integer.MIN_VALUE) {
                    // 当天获利当天结算，动态结
算
                    dp[i][j] = Math.max(dp[i]
[j],dp[i-1][j-1] +prices[i-1] -prices[i-2]);
                    // 要么等于前一天的同一阶段
(dp[i][j])，要么进行阶段转移，计算获利
                }
            }
            print(dp);
            // 阶段2, 4: dp[i][j] = max{dp[i-
1][j] + prices[i-1] - prices[i-2],dp[i-1][j-
1]}
            for(int j = 2;j <= 5;j += 2 ) {
                dp[i][j] =dp[i-1][j-1]; // 先
初始化为前一天的上一个状态
                if(i > 1 &&dp[i-1][j] !=
Integer.MIN_VALUE ) {
                    dp[i][j] = Math.max(dp[i]
[j],dp[i-1][j] +prices[i-1] -prices[i-2]);
                    // 要么继续持有股票(dp[i]
[j]),要么把股票卖掉
                }
            }
            print(dp);
        }

        int res = 0;
        for(int j = 1;j <= 5;j += 2 ) {
            res = Math.max(res,dp[N][j]);
```

```
        }

        return res;
    }

    private void print(int[][] matrix) {
        for( int i = 0; i < matrix.length; i++ ) {

System.out.println(Arrays.toString(matrix[i]));
        }
        System.out.println();
    }
}
```

如果我可以买卖K次

188. Best Time to Buy and Sell Stock IV

难度 困难 122 收藏 分享 切换为中文 关注

题目描述 评论 (75) 题解(15) New 提交记录

Say you have an array for which the *i*-th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most **k** transactions.

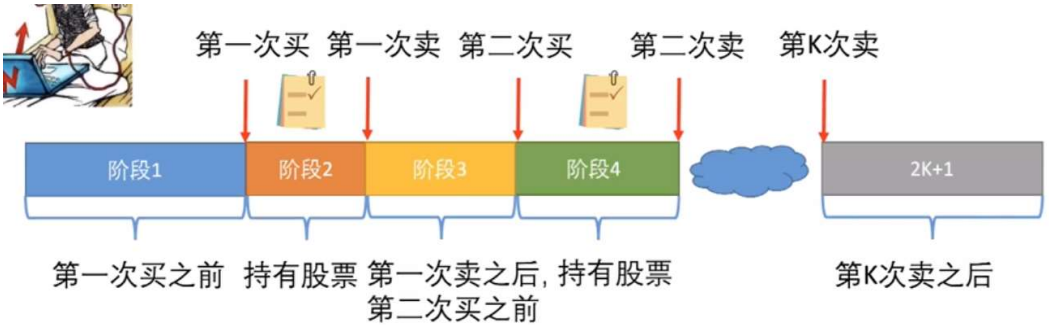
Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Example 1:

```
Input: [2,4,1], k = 2
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit
= 4-2 = 2.
```

Example 2:

```
Input: [3,2,6,5,0,3], k = 2
Output: 7
Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit
= 6-2 = 4.
             Then buy on day 5 (price = 0) and sell on day 6 (price = 3),
profit = 3-0 = 3.
```




```java
class Solution {
    public int maxProfit(int k, int[] prices) {
        int N = prices.length;
        if( N == 0 ) {
            return 0;
        }
        if( k > N/2 ) {
            // 直接贪心
            int profit = 0;
            for( int i = 1; i < prices.length; i++ ) {
                if( prices[i] > prices[i-1] ) {
                    profit += prices[i] - prices[i-1];
                }
            }
            return profit;
        }
        int[][] dp = new int[N+1][2*k+1+1];
        for( int i = 0; i <= 2*k+1; i++ ) {
            dp[0][i] = Integer.MIN_VALUE;
        }
        dp[0][1] = 0;
        for( int i = 1; i <= N; i++ ) {
            // 阶段1，3，5：  dp[i][j] = max{dp[i-1], dp[i-1][j-1] + prices[i-1] - prices[i-2]}
            for( int j = 1; j <= 2*k+1; j += 2 ) {
                dp[i][j] = dp[i-1][j];
                // sell
                if( j > 1 && i > 1 && dp[i-1][j-1] != Integer.MIN_VALUE) {
                    dp[i][j] = Math.max(dp[i][j], dp[i-1][j-1] + prices[i-1] - prices[i-2]);
                }
            }
            // 阶段2，4：  dp[i][j] = max{dp[i-1][j] + prices[i-1] - prices[i-2], dp[i-1][j-1]}
            for( int j = 2; j <= 2*k+1; j += 2 ) {
                dp[i][j] = dp[i-1][j-1];
                if( i > 1 && dp[i-1][j] != Integer.MIN_VALUE ) {
                    dp[i][j] = Math.max(dp[i][j], dp[i-1][j] + prices[i-1] - prices[i-2]);
                }
            }
        }

        int res = 0;
        for( int j = 1; j <= 2*k+1; j += 2 ) {
            res = Math.max(res, dp[N][j]);
        }
        return res;
    }
}
```

最长上升子序列 Longest Increasing Subsequence

300. Longest Increasing Subsequence

难度 中等 329 收藏 分享 切换为中文 关注

题目描述 评论 (168) 题解(52) New 提交记录

Given an unsorted array of integers, find the length of longest increasing subsequence.

Example:

```
Input: [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is
[2,3,7,101], therefore the length is 4.
```

Note:

- There may be more than one LIS combination, it is only necessary for you to return the length.
- Your algorithm should run in $O(n^2)$ complexity.

Follow up: Could you improve it to $O(n \log n)$ time complexity?

300. Longest Increasing Subsequence

难度 中等 329 收藏 分享 切换为中文 关注

题目描述 评论 (168) 题解(52) New 提交记录

执行结果: 通过 显示详情 >

执行用时: 29 ms, 在所有 java 提交中击败了 50.58% 的用户

内存消耗: 36.7 MB, 在所有 java 提交中击败了 30.29% 的用户

炫耀一下:

进行下一个挑战:

递增的三元子序列	中等
俄罗斯套娃信封问题	困难
最长数对链	中等
最长递增子序列的个数	中等
两个字符串的最小ASCII删除和	中等

提交时间	提交结果	执行用时	内存消耗	语言

Java

```java
class Solution {
    public int lengthOfLIS(int[] nums) {
        int N = nums.length;
        if( N == 0 ) {
            return 0;
        }
        int[] dp = new int[N];
        dp[0] = 1;
        // dp[i] 指前i个数最长上升子序列的长度
        int ans = 1;
        for( int i = 1; i < N; i++ ) {
            dp[i] = 1;
            for( int j = 0; j < N; j++ ) {
                // 看看比之前dp[]中哪个数要大
                // 选刚好比当前nums[i]小的来更新长度
                if( nums[i] > nums[j] ) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            ans = Math.max(ans, dp[i]);
        }
        return ans;
    }
}
```