122. Best Time to Buy and Sell Stock II

July 12, 2016 | 471.4K views

Say you have an array `prices` for which the ith 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-
```

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 i
             engaging multiple transactions at the same time. You must sell before buy
```

Example 3:

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

Constraints:

- `1 <= prices.length <= 3 * 10 ^ 4`
- `0 <= prices[i] <= 10 ^ 4`

Summary

We have to determine the maximum profit that can be obtained by making the transactions (no limit on the number of transactions done). For this we need to find out those sets of buying and selling prices which together lead to the maximization of profit.

Solution

Approach 1: Brute Force

In this case, we simply calculate the profit corresponding to all the possible sets of transactions and find out the maximum profit out of them.

Java

```
class Solution {
    public int maxProfit(int[] prices) {
        return calculate(prices, 0);
    }

    public int calculate(int prices[], int s) {
        if (s >= prices.length)
            return 0;
        int max = 0;
        for (int start = s; start < prices.length; start++) {
            int maxprofit = 0;
            for (int i = start + 1; i < prices.length; i++) {
                if (prices[start] > prices[i]) {
                    int profit = calculate(prices, i + 1) + prices[i] - prices[start];
                    if (profit > maxprofit)
                        maxprofit = profit;
                }
            }
            if (maxprofit > max)
                max = maxprofit;
        }
        return max;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^n)$. Recursive function is called n^n times.
- Space complexity : $O(n)$. Depth of recursion is n.

Approach 2: Peak Valley Approach

Algorithm

Say the given array is:

[7, 1, 5, 3, 6, 4].

If we plot the numbers of the given array on a graph, we get:

If we analyze the graph, we notice that the points of interest are the consecutive valleys and peaks.

Mathematically speaking:

$$TotalProfit = \sum_i (height(peak_i) - height(valley_i))$$

The key point is we need to consider every peak immediately following a valley to maximize the profit. In case we skip one of the peaks (trying to obtain more profit), we will end up losing the profit over one of the transactions leading to an overall lesser profit.

For example, in the above case, if we skip peak$_0$ and valley$_1$ trying to obtain more profit by considering points with more difference in heights, the net profit obtained will always be lesser than the one obtained by including them, since C will always be lesser than $A + B$.

Java

```
class Solution {
    public int maxProfit(int[] prices) {
        int i = 0;
        int valley = prices[0];
        int peak = prices[0];
        int maxprofit = 0;
        while (i < prices.length - 1) {
            while (i < prices.length - 1 && prices[i] >= prices[i + 1])
                i++;
            valley = prices[i];
            while (i < prices.length - 1 && prices[i] <= prices[i + 1])
                i++;
            peak = prices[i];
            maxprofit += peak - valley;
        }
        return maxprofit;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Single pass.
- Space complexity : $O(1)$. Constant space required.

Approach 3: Simple One Pass

Algorithm

This solution follows the logic used in Approach 2 itself, but with only a slight variation. In this case, instead of looking for every peak following a valley, we can simply go on crawling over the slope and keep on adding the profit obtained from every consecutive transaction. In the end,we will be using the peaks and valleys effectively, but we need not track the costs corresponding to the peaks and valleys along with the maximum profit, but we can directly keep on adding the difference between the consecutive numbers of the array if the second number is larger than the first one, and at the total sum we obtain will be the maximum profit. This approach will simplify the solution. This can be made clearer by taking this example:

[1, 7, 2, 3, 6, 7, 6, 7]

The graph corresponding to this array is:

From the above graph, we can observe that the sum $A + B + C$ is equal to the difference D corresponding to the difference between the heights of the consecutive peak and valley.

Java

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

Complexity Analysis

- Time complexity : $O(n)$. Single pass.
- Space complexity: $O(1)$. Constant space needed.

Rate this article:

Previous Next