<https://leetcode.com/problems/best-time-to-buy-and-sell-stock/>

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

**Example 1:**

Input: prices = [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.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

**Example 2:**

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

**Constraints:**

1 <= prices.length <= 10^5

0 <= prices[i] <= 10^4

**Attempt 1: 2023-03-30**

**Solution 1:  Brute Force (10 min, TLE)**

class Solution {

    public int maxProfit(int[] prices) {

        int n = prices.length;

        int maxProfit = 0;

        for(int i = 0; i < n; i++) {

            for(int j = i + 1; j < n; j++) {

                maxProfit = Math.max(maxProfit, prices[j] - prices[i]);

            }

        }

        return maxProfit;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock/editorial/>

public class Solution {

    public int maxProfit(int prices[]) {

        int maxprofit = 0;

        for (int i = 0; i < prices.length - 1; i++) {

            for (int j = i + 1; j < prices.length; j++) {

                int profit = prices[j] - prices[i];

                if (profit > maxprofit)

                    maxprofit = profit;

            }

        }

        return maxprofit;

    }

}

**Complexity Analysis**

Time complexity: O(n^2). Loop runs n \* (n−1) / 2 times.

Space complexity: O(1). Only two variables - maxprofit and profit are used.

**Solution 2: One Pass (10 min)**

class Solution {

    public int maxProfit(int[] prices) {

        int minVal = Integer.MAX\_VALUE;

        int maxProfit = 0;

        for(int i = 0; i < prices.length; i++) {

            if(prices[i] < minVal) {

                minVal = prices[i];

            } else if(prices[i] - minVal > maxProfit) {

                maxProfit = prices[i] - minVal;

            }

        }

        return maxProfit;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock/editorial/>

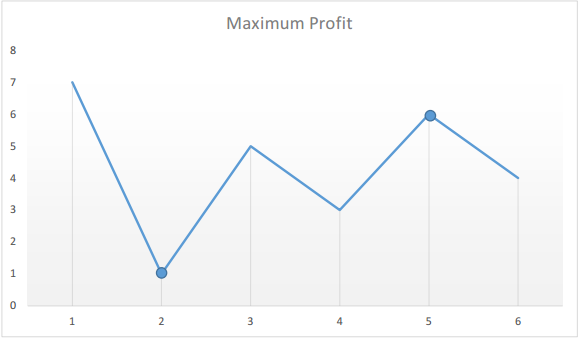
**Approach 2: One Pass**

**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:



The points of interest are the peaks and valleys in the given graph. We need to find the largest price following each valley, which difference could be the max profit. We can maintain two variables - minprice and maxprofit corresponding to the smallest valley and maximum profit (maximum difference between selling price and minprice) obtained so far respectively.

public class Solution {

    public int maxProfit(int prices[]) {

        int minprice = Integer.MAX\_VALUE;

        int maxprofit = 0;

        for (int i = 0; i < prices.length; i++) {

            if (prices[i] < minprice)

                minprice = prices[i];

            else if (prices[i] - minprice > maxprofit)

                maxprofit = prices[i] - minprice;

        }

        return maxprofit;

    }

}

**Complexity Analysis**

Time complexity: O(n). Only a single pass is needed.

Space complexity: O(1). Only two variables are used.

**Solution 3: 1D DP and Kadane's Algorithm (30 min)**

L121.Best Time to Buy and Sell Stock 1D DP solution

class Solution {

    public int maxProfit(int[] prices) {

        int n = prices.length;

        int[] diff = new int[n];

        diff[0] = 0;

        for(int i = 1; i < n; i++) {

            diff[i] = prices[i] - prices[i - 1];

        }

        int[] dp = new int[n];

        dp[0] = diff[0];

        int max = dp[0];

        for(int i = 1; i < n; i++) {

            dp[i] = (dp[i - 1] > 0 ? dp[i - 1] : 0) + diff[i];

            max = Math.max(max, dp[i]);

        }

        return max;

    }

}

================================================

L121.Best Time to Buy and Sell Stock Kadane's Algorithm solution

class Solution {

    public int maxProfit(int[] prices) {

        int n = prices.length;

        int[] diff = new int[n];

        diff[0] = 0;

        for(int i = 1; i < n; i++) {

            diff[i] = prices[i] - prices[i - 1];

        }

        int sum = diff[0];

        int max = diff[0];

        for(int i = 1; i < n; i++) {

            sum = sum < 0 ? diff[i] : sum + diff[i];

            max = Math.max(max, sum);

        }

        return max;

    }

}

**Key: Modify original input into adjacent numbers difference sequence, which can apply 1D DP related Kadane's Algorithm, which is exactly same as L53.Maximum Subarray**

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock/solutions/39038/kadane-s-algorithm-since-no-one-has-mentioned-about-this-so-far-in-case-if-interviewer-twists-the-input/>

The logic to solve this problem is same as "max subarray problem" using Kadane's Algorithm. Since no body has mentioned this so far, I thought it's a good thing for everybody to know.

L53.Maximum Subarray 1D DP solution

class Solution {

    // dp[i] denotes maximum subarray sum ending at i (including nums[i])

    // maxSubArray(int nums[], int i), which means the maxSubArray for nums[0:i] which must has A[i] as the end element.

    // maxSubArray(nums, i) = maxSubArray(nums, i - 1) > 0 ? maxSubArray(nums, i - 1) : 0 + nums[i];

    public int maxSubArray(int[] nums) {

        int[] dp = new int[nums.length];

        dp[0] = nums[0];

        int max = dp[0];

        for(int i = 1; i < nums.length; i++) {

            dp[i] = (dp[i - 1] > 0 ? dp[i - 1] : 0) + nums[i];

            max = Math.max(max, dp[i]);

        }

        return max;

    }

}

================================================

L53.Maximum Subarray Kadane's algorithm solution

class Solution {

    public int maxSubArray(int[] nums) {

        // Replace dp[i] with 'sum'

        int sum = nums[0];

        int max = nums[0];

        for(int i = 1; i < nums.length; i++) {

            sum = sum < 0 ? nums[i] : sum + nums[i];

            max = Math.max(max, sum);

        }

        return max;

    }

}

All the straight forward solution should work, but if the interviewer twists the question slightly by giving the ***difference array of prices***, Ex: for

{1, 7, 4, 11}, if he gives {0, 6, -3, 7}, you might end up being confused.

Here, the logic is to calculate the difference (maxCur += prices[i] - prices[i-1]) of the original array, and find a contiguous subarray giving maximum profit. If the difference falls below 0, reset it to zero.

    public int maxProfit(int[] prices) {

        int maxCur = 0, maxSoFar = 0;

        for(int i = 1; i < prices.length; i++) {

            maxCur = Math.max(0, maxCur += prices[i] - prices[i-1]);

            maxSoFar = Math.max(maxCur, maxSoFar);

        }

        return maxSoFar;

    }\*

maxCur = current maximum value\*

maxSoFar = maximum value found so far

**Refer to**

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)