Writing programming interview questions hasn't made me rich. Maybe trading Apple stocks will.

Suppose we could access yesterday's stock prices as an array, where:

- The indices are the time in minutes past trade opening time, which was 9:30am local time.
- The values are the price in dollars of Apple stock at that time.

So if the stock cost $500 at 10:30am, `stockPricesYesterday[60] = 500`.

Write an efficient function that takes `stockPricesYesterday` and returns **the best profit I could have made from 1 purchase and 1 sale of 1 Apple stock yesterday.**

For example:

JavaScript
```
var stockPricesYesterday = [10, 7, 5, 8, 11, 9];

getMaxProfit(stockPricesYesterday);
// returns 6 (buying for $5 and selling for $11)
```

No "shorting"—you must buy before you sell. You may not buy *and* sell in the same time step (at least 1 minute must pass).

Gotchas

It is not sufficient to simply take the difference between the highest price and the lowest price, because the highest price may come *before* the lowest price. You must buy before you sell.

What if the stock value *goes down all day*? In that case, the best profit will be **negative**.

You can do this in $O(n)$ time and $O(1)$ space!

Breakdown

To start, try writing an example value for `stockPricesYesterday` and finding the maximum profit "by hand." What's your process for figuring out the maximum profit?

The brute force approach would be to try *every pair of times* (treating the earlier time as the buy time and the later time as the sell time) and see which one is higher.

```
/*
In JavaScript
```

JavaScript

```
function getMaxProfit(stockPricesYesterday) {

    var maxProfit = 0;

    // go through every time
    for (var outerTime = 0; outerTime < stockPricesYesterday.length; outerTime++) {

        // for every time, got through every OTHER time
        for (var innerTime = 0; innerTime < stockPricesYesterday.length; innerTime++) {

            // for each pair, find the earlier and later times
            var earlierTime = Math.min(outerTime, innerTime);
            var laterTime   = Math.max(outerTime, innerTime);

            // and use those to find the earlier and later prices
            var earlierPrice = stockPricesYesterday[earlierTime];
            var laterPrice   = stockPricesYesterday[laterTime];

            // see what our profit would be if we bought at the
            // min price and sold at the current price
            var potentialProfit = laterPrice - earlierPrice;

            // update maxProfit if we can do better
            maxProfit = Math.max(maxProfit, potentialProfit);
        }
    }

    return maxProfit;
}
```

But that will take $O(n^2)$ time, since we have two nested loops—for *every* time, we're going through *every other* time. Can we do better?

Well, we’re doing a lot of extra work. We’re looking at every pair *twice*. We know we have to buy before we sell, so in our *inner for loop* we could just look at every price **after** the price in our *outer for loop*.

That could look like this:

```javascript
function getMaxProfit(stockPricesYesterday) {

    var maxProfit = 0;

    // go through every price and time
    for (var earlierTime = 0; earlierTime < stockPricesYesterday.length; earlierTime++) {
        var earlierPrice = stockPricesYesterday[earlierTime];

        // and go through all the LATER prices
        for (var laterTime = earlierTime + 1; laterTime < stockPricesYesterday.length; laterTime++)
            var laterPrice = stockPricesYesterday[laterTime];

            // see what our profit would be if we bought at the
            // min price and sold at the current price
            var potentialProfit = laterPrice - earlierPrice;

            // update maxProfit if we can do better
            maxProfit = Math.max(maxProfit, potentialProfit);
        }
    }

    return maxProfit;
}
```

What's our runtime now?

Well, our outer for loop goes through *all* the times and prices, but our inner for loop goes through *one fewer price each time*. So our total number of steps is the sum $n + (n-1) + (n-2) ... + 2 + 1$, which is *still* $O(n^2)$ time.

We can do better!

If we're going to do better than $O(n^2)$, we're probably going to do it in either $O(n \lg n)$ or $O(n)$. $O(n \lg n)$ comes up in sorting and searching algorithms where we're recursively cutting the set in half. It's not obvious that we can save time by cutting the set in half here. Let's first see how well we can do by looping through the set only *once*.

Since we're trying to loop through the set once, let's use a greedy approach, where we keep a running `maxProfit` until we reach the end. We'll start our `maxProfit` at $0. As we're iterating, how do we know if we've found a new `maxProfit`?

At each iteration, our `maxProfit` is either:

1. the same as the `maxProfit` at the last time step, or
2. the max profit we can get by selling at the `currentPrice`

How do we know when we have case (2)?

The max profit we can get by selling at the `currentPrice` is simply the difference between the `currentPrice` and the `minPrice` from earlier in the day. If this difference is greater than the current `maxProfit`, we have a new `maxProfit`.

So for every price, we'll need to:

- keep track of the **lowest price we've seen so far**
- see if we can get a **better profit**

Here's one possible solution:

JavaScript

```
function getMaxProfit(stockPricesYesterday) {

    var minPrice = stockPricesYesterday[0];
    var maxProfit = 0;

    for (var i = 0; i < stockPricesYesterday.length; i++) {
        var currentPrice = stockPricesYesterday[i];

        // ensure min_price is the lowest price we've seen so far
        minPrice = Math.min(minPrice, currentPrice);

        // see what our profit would be if we bought at the
        // min price and sold at the current price
        var potentialProfit = currentPrice - minPrice;

        // update maxProfit if we can do better
        maxProfit = Math.max(maxProfit, potentialProfit);
    }

    return maxProfit;
}
```

We're finding the max profit with one pass and constant space!

Are we done? Let's think about some edge cases. What if the stock value *stays the same*? What if the stock value *goes down all day*?

If the stock price doesn't change, the max possible profit is 0. Our function will correctly return that. So we're good.

But if the value *goes down all day*, we're in trouble. Our function would return 0, but there's no way we could break even if the price always goes down.

How can we handle this?

Well, what are our options? Leaving our function as it is and just returning zero is *not* a reasonable option—we wouldn't know if our best profit was negative or *actually* zero, so we'd be losing information. Two reasonable options could be:

1. **return a negative profit**. "What's the least badly we could have done?"
2. **throw an error**. "We should not have purchased stocks yesterday!"

In this case, it's probably best to go with option (1). The advantages of returning a negative profit are:

- We **more accurately answer the challenge**. If profit is "revenue minus expenses", we're returning the *best* we could have done.
- It's **less opinionated**. We'll leave decisions up to our function's users. It would be easy to wrap our function in a helper function to decide if it's worth making a purchase.
- We allow ourselves to **collect better data**. It *matters* if we would have lost money, and it *matters* how much we would have lost. If we're trying to get rich, we'll probably care about those numbers.

How can we adjust our function to return a negative profit if we can only lose money? Initializing `maxProfit` to 0 won't work...

Well, we started our `minPrice` at the first price, so let's start our `maxProfit` at the *first profit we could get*—if we buy at the first time and sell at the second time.

```javascript
minPrice = stockPricesYesterday[0]
maxProfit = stockPricesYesterday[1] - stockPricesYesterday[0]
```

But we have the potential for an index out of bounds error here, if `stockPricesYesterday` has fewer than 2 prices.

We *do* want to throw an error in that case, since *profit* requires buying *and* selling, which we can't do with less than 2 prices. So rather than throwing a confusing index out of bounds error, let's explicitly catch that case and throw a more helpful error message:

```javascript
if (stockPricesYesterday.length < 2) {
    throw new Error('Getting a profit requires at least 2 prices');
}

var minPrice = stockPricesYesterday[0];
var maxProfit = stockPricesYesterday[1] - stockPricesYesterday[0];
```

Ok, does that work?

No! **`maxProfit` is still always 0!** What's happening?

If the price always goes down, `minPrice` is always set to the `currentPrice`. So `currentPrice - minPrice` comes out to 0, which of course will always be greater than a negative profit.

When we're calculating the `maxProfit`, we need to make sure we never have a case where we try **both buying and selling stocks at the `currentPrice`**.

To make sure we're always buying at an *earlier* price, never the `currentPrice`, let's switch the order around so we calculate `maxProfit` *before* we update `minPrice`.

We'll also need to pay special attention to time 0. Make sure we don't try to buy *and* sell at time 0!

Solution

We'll greedily walk through the array to track the max profit and lowest price so far.

For every price, we check if:

- we can get a better profit by buying at `minPrice` and selling at the `currentPrice`
- we have a new `minPrice`

To start, we initialize:

1. `minPrice` as the first price of the day
2. `maxProfit` as the first profit we could get

We decided to return a *negative* profit if the price decreases all day and we can't make any money. We could have thrown an error instead, but returning the negative profit is cleaner, makes our function less opinionated, and ensures we don't lose information.

JavaScript

```
function getMaxProfit(stockPricesYesterday) {

    // make sure we have at least 2 prices
    if (stockPricesYesterday.length < 2) {
        throw new Error('Getting a profit requires at least 2 prices');
    }

    // we'll greedily update minPrice and maxProfit, so we initialize
    // them to the first price and the first possible profit
    var minPrice = stockPricesYesterday[0];
    var maxProfit = stockPricesYesterday[1] - stockPricesYesterday[0];

    // start at the second (index 1) time
    // we can't sell at the first time, since we must buy first,
    // and we can't buy and sell at the same time!
    // if we started at index 0, we'd try to buy /and/ sell at time 0.
    // this would give a profit of 0, which is a problem if our
    // maxProfit is supposed to be /negative/--we'd return 0!
    for (var i = 1; i < stockPricesYesterday.length; i++) {
        var currentPrice = stockPricesYesterday[i];

        // see what our profit would be if we bought at the
        // min price and sold at the current price
        var potentialProfit = currentPrice - minPrice;

        // update maxProfit if we can do better
        maxProfit = Math.max(maxProfit, potentialProfit);

        // update minPrice so it's always
        // the lowest price we've seen so far
        minPrice = Math.min(minPrice, currentPrice);
    }

    return maxProfit;
}
```

Complexity

$O(n)$ time and $O(1)$ space. We only loop through the array once.

Did you get it right?

Yes, I'm expert on this | Not quite, review later

Like this problem? Pass it on!

(https://www.facebook.com/InterviewCake)(https://twitter.com/interviewcake)

Subscribe to our weekly question email list » (/free-weekly-coding-interview-problem-newsletter)

Programming interview questions by company:

- Google interview questions (/google-interview-questions)
- Facebook interview questions (/facebook-interview-questions)
- Amazon interview questions (/amazon-interview-questions)

Programming interview questions by language:

- Java interview questions (/java-interview-questions)
- Python interview questions (/python-interview-questions)
- Ruby interview questions (/ruby-interview-questions)
- JavaScript interview questions (/javascript-interview-questions)
- **NEW:** Testing and QA interview questions (/testing-and-qa-interview-questions)
- **NEW:** SQL interview questions (/sql-interview-questions)

228 Park Ave S #82632, New York, NY US 10003 (804) 876-2253