Interview Cake

Writing programming interview questions hasn't made me rich yet ... so I might give up and start trading Apple stocks all day instead.

First, I wanna know how much money I *could have* made yesterday if I'd been trading Apple stocks all day.

So I grabbed Apple's stock prices from yesterday and put them in an array called `stockPrices`, where:

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

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

Write an efficient method that takes `stockPrices` and returns **the best profit I could have made from one purchase and one sale of one share of Apple stock yesterday.**

For example:

```java
int[] stockPrices = new int[] {10, 7, 5, 8, 11, 9};

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

No "shorting"—you need to buy before you can sell. Also, you can't buy *and* sell in the same time step—at least 1 minute has to pass.

Gotchas

You can't just take the difference between the highest price and the lowest price, because the highest price might come *before* the lowest price. And you have to buy before you can sell.

What if the price *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 `stockPrices` 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.

Java

```
public static int getMaxProfit(int[] stockPrices) {

    int maxProfit = 0;

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

        // for every time, go through every other time
        for (int innerTime = 0; innerTime < stockPrices.length; innerTime++) {

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

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

            // see what our profit would be if we bought at the
            // min price and sold at the current price
            int 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. Also, **it's not correct**: we won't ever report a negative profit! 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:

Java

```
public static int getMaxProfit(int[] stockPrices) {

    int maxProfit = 0;

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

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

            // see what our profit would be if we bought at the
            // min price and sold at the current price
            int 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 array in half. It's not obvious that we can save time by cutting the array in half here. Let's first see how well we can do by looping through the array only *once*.

Since we're trying to loop through the array 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:

Java

```
public static int getMaxProfit(int[] stockPrices) {

    int minPrice = stockPrices[0];
    int maxProfit = 0;

    for (int currentPrice : stockPrices) {

        // ensure minPrice 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
        int 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 price *stays the same*? What if the price *goes down all day*?

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

But if the value *goes down all day*, we're in trouble. Our method 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 method 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 exception**. "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 method's users. It would be easy to wrap our method in a helper method 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 method 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.

```java
minPrice = stockPrices[0];
maxProfit = stockPrices[1] - stockPrices[0];
```

But we have the potential for an `ArrayIndexOutOfBoundsException` here, if `stockPrices` has fewer than 2 prices.

We *do* want to throw an exception in that case, since *profit* requires buying *and* selling, which we can't do with less than 2 prices. So, let's explicitly check for this case and handle it:

```java
if (stockPrices.length < 2) {
    throw new IllegalArgumentException("Getting a profit requires at least 2 prices");
}

int minPrice = stockPrices[0];
int maxProfit = stockPrices[1] - stockPrices[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 exception instead, but returning the negative profit is cleaner, makes our method less opinionated, and ensures we don't lose information.

Java

```
public static int getMaxProfit(int[] stockPrices) {

    if (stockPrices.length < 2) {
        throw new IllegalArgumentException("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
    int minPrice = stockPrices[0];
    int maxProfit = stockPrices[1] - stockPrices[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 (int i = 1; i < stockPrices.length; i++) {
        int currentPrice = stockPrices[i];

        // see what our profit would be if we bought at the
        // min price and sold at the current price
        int 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.

What We Learned

This one's a good example of the greedy approach in action. Greedy approaches are great because they're *fast* (usually just one pass through the input). But they don't work for every problem.

How do you know if a problem will lend itself to a greedy approach? Best bet is to try it out and see if it works. Trying out a greedy approach should be one of the first ways you try to break down a new question.

To try it on a new problem, start by asking yourself:

"Suppose we *could* come up with the answer in one pass through the input, by simply updating the 'best answer so far' as we went. What ***additional values*** would we need to keep updated as we looked at each item in our input, in order to be able to update the **'best answer so far'** in constant time?"

In *this* case:

The "**best answer so far**" is, of course, the max profit that we can get based on the prices we've seen so far.

The "**additional value**" is the minimum price we've seen so far. If we keep that updated, we can use it to calculate the new max profit so far in constant time. The max profit is the larger of:

1. The previous max profit
2. The max profit we can get by selling now (the current price minus the minimum price seen so far)

Try applying this greedy methodology to future questions.