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 function 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:

```
const stockPrices = [10, 7, 5, 8, 11, 9];


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

JavaScript

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!↴

> Not sure what this is? It's **big O notation**—a tool we use for talking about how much time an algorithm takes to run or how much memory a data structure takes up in our computer.
>
> It's pretty simple. Learn about big O notation ➡ (/big-o-notation-time-and-space-complexity)

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↴

> A **brute force** algorithm finds a solution by trying *all* possible answers and picking the best one.
>
> Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?
>
> You could try running through all potential coin combinations and pick the one that adds to 67 cents using the fewest coins. That's a *brute force* algorithm, since you're trying *all* possible ways to make change.
>
> Here are a few other brute force algorithms:
>
> - Trying to fit as many overlapping meetings as possible in a conference room? Run through all possible schedules, and pick the schedule that fits the most meetings in the room.
> - Trying to find the cheapest route through a set of cities? Try all possible routes and pick the cheapest one.
> - Looking for a minimum spanning tree in a graph (/concept/graph)? Try all possible sets of edges, and pick the cheapest set that's also a tree.

Brute force solutions are usually *very slow* since they involve testing a huge number of possible answers.

Brute force approaches are rarely the most efficient. Other approaches, like greedy algorithms (/concept/greedy) or dynamic programming (/concept/bottom-up) tend to be faster.

Even so, talking through a brute force solution can be a good first step in a coding interview. It's usually pretty easy to derive, so it allows you to quickly make progress and come up with *something* that works. From there, you have some helpful boundaries for refining your algorithm—you're only interested in solutions that are faster (and/or more space efficient) than the brute force solution you've already come up with.

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.

JavaScript

```
function getMaxProfit(stockPrices) {
  let maxProfit = 0;

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

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

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

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

      // See what our profit would be if we bought at the
      // min price and sold at the current price
      const 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,

> Not sure what this is? It's **big O notation**—a tool we use for talking about how much time an algorithm takes to run or how much memory a data structure takes up in our computer.
>
> It's pretty simple. Learn about big O notation ➡ (/big-o-notation-time-and-space-complexity)

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:

```javascript
function getMaxProfit(stockPrices) {
  let maxProfit = 0;

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

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

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

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

  return maxProfit;
}
```

JavaScript

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 $n + (n-1) + (n-2) ... + 2 + 1$

The **sum of integers** $1..n$ is $\approx \frac{n^2}{2}$, which is $O(n^2)$

Series like this actually come up quite a bit:

$$1 + 2 + 3 + ... + (n-1) + n$$

Or, equivalently, the other way around:

$$n + (n-1) + ... + 3 + 2 + 1$$

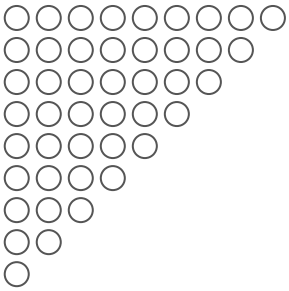
And sometimes the last n is omitted, but as we'll see it doesn't affect the big O:

$$(n-1) + (n-2) + ... + 3 + 2 + 1$$

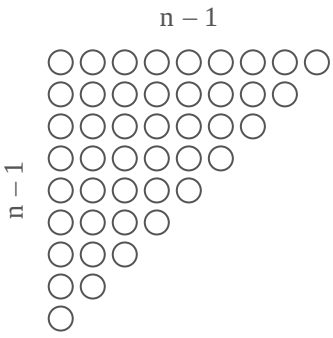
Let's draw this out. Let's say $n = 10$, so we'll represent $n - 1$ as nine circles:

We can continue the pattern with $n - 2$

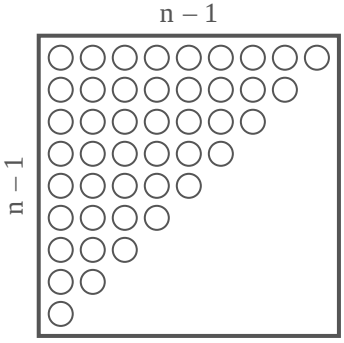
And $n - 3$, $n - 4$, etc, all the way down to 1:



Notice both the top and right "sides" of our set of circles have $n - 1$ items:



In fact, we could imagine our circles inside of a square with sides of length n-1:



Notice that we've filled in just about half of the square!

Of course, the area of the square is $(n-1) * (n-1)$, which is $O(n^2)$. Our total number of circles is about half of that, so $O(n^2/2)$, which is still $O(n^2)$. Remember: with big O notation (/big-o-notation-time-and-space-complexity), we throw out the constants.

If we had started from n instead of $n-1$ we'd have $O(n^2 + n)$, which again is still $O(n^2)$ since in big O notation we drop the less significant terms.

, 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:

JavaScript

```
function getMaxProfit(stockPrices) {
  let minPrice = stockPrices[0];
  let maxProfit = 0;

  for (let i = 0; i < stockPrices.length; i++) {
    const currentPrice = stockPrices[i];

    // 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
    const 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 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 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 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.

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

But we have the potential for reading undefined values 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:

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

let minPrice = stockPrices[0];
let maxProfit = stockPrices[1] - stockPrices[0];
```
JavaScript

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 function less opinionated, and ensures we don't lose information.

JavaScript

```
function getMaxProfit(stockPrices) {
  if (stockPrices.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
  let minPrice = stockPrices[0];
  let 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 (let i = 1; i < stockPrices.length; i++) {
    const currentPrice = stockPrices[i];

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

Not sure what this is? It's **big O notation**—a tool we use for talking about how much time an algorithm takes to run or how much memory a data structure takes up in our computer.

It's pretty simple. Learn about big O notation ➡ (/big-o-notation-time-and-space-complexity)

We only loop through the array once.

What We Learned

This one's a good example of the greedy↴

A **greedy** algorithm builds up a solution by choosing the option that looks the best at every step.

Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?

Whenever picking which coin to use, you'd take the highest-value coin you could. A quarter, another quarter, then a dime, a nickel, and finally two pennies. That's a *greedy* algorithm, because you're always *greedily* choosing the coin that covers the biggest portion of the remaining amount.

Some other places where a greedy algorithm gets you the best solution:

- Trying to fit as many overlapping meetings as possible in a conference room? At each step, schedule the meeting that *ends* earliest.
- Looking for a minimum spanning tree in a graph (/concept/graph)? At each step, greedily pick the cheapest edge that reaches a new vertex.

Careful: sometimes a greedy algorithm *doesn't* give you an optimal solution:

- When filling a duffel bag with cakes of different weights and values (/question/cake-thief), choosing the cake with the highest value per pound doesn't always produce the best haul.
- To find the cheapest route visiting a set of cities, choosing to visit the the cheapest city you haven't been to yet doesn't produce the cheapest overall itinerary.

Validating that a greedy strategy always gets the best answer is tricky. Either prove that the answer produced by the greedy algorithm is as good as an optimal answer, or run through a rigorous set of test cases to convince your interviewer (and yourself) that its correct.

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.