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, `stock_prices_yesterday[60] = 500`.

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

For example:

```
Ruby

stock_prices_yesterday = [10, 7, 5, 8, 11, 9]

get_max_profit(stock_prices_yesterday)
# 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 `stock_prices_yesterday` and finding the maximum profit "by hand." What's your process for figuring out the maximum profit?

The brute force

> A **brute force** algorithm simply tries *all* possible answers to a question and checks them for correctness.
>
> It's rarely the most efficient approach, but it can be helpful to consider the time cost of the brute force approach when building an optimized solution. If your solution isn't faster than the brute force approach, it may not be optimal.

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.

Ruby

```ruby
def get_max_profit(stock_prices_yesterday)

    max_profit = 0

    # go through every time
    for outer_time in (0...stock_prices_yesterday.length)

        # for every time, go through every OTHER time
        for inner_time in (0...stock_prices_yesterday.length)

            # for each pair, find the earlier and later times
            earlier_time = [outer_time, inner_time].min
            later_time   = [outer_time, inner_time].max

            # and use those to find the earlier and later prices
            earlier_price = stock_prices_yesterday[earlier_time]
            later_price   = stock_prices_yesterday[later_time]

            # see what our profit would be if we bought at the
            # earlier price and sold at the later price
            potential_profit = later_price - earlier_price

            # update max_profit if we can do better
            max_profit = [max_profit, potential_profit].max
        end
    end

    return max_profit
end
```

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:

Ruby

```
def get_max_profit(stock_prices_yesterday)

    max_profit = 0

    # go through every price (with it's index as the time)
    stock_prices_yesterday.each_with_index do |earlier_price, earlier_time|

        # and go through all the LATER prices
        for later_price in stock_prices_yesterday[earlier_time+1..-1]

            # see what our profit would be if we bought at the
            # earlier price and sold at the later price
            potential_profit = later_price - earlier_price

            # update max_profit if we can do better
            max_profit = [max_profit, potential_profit].max
        end
    end

    return max_profit
end
```

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:

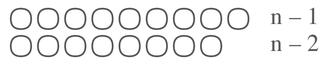
$$n + (n-1) + ... + 3 + 2 + 1$$

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

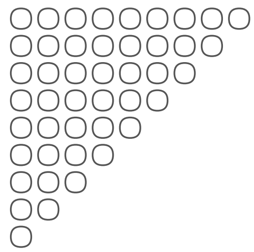
$$(n-1) + (n-2) + ... + 3 + 2 + 1$$

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

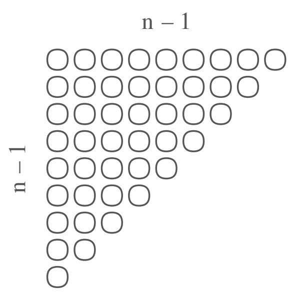
We can continue the pattern with $n - 2$



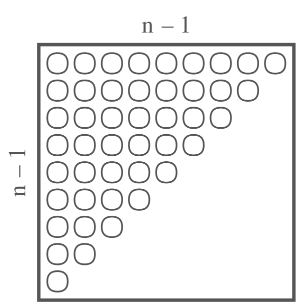
And $n - 3$, $n - 4$, etc, all the way down to 1:



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 set once, let's use a greedy

> A **greedy** algorithm iterates through the problem space taking the optimal solution "so far," until it reaches the end.
>
> The greedy approach is only optimal if the problem has "optimal substructure," which means stitching together optimal solutions to subproblems yields an optimal solution.

approach, where we keep a running `max_profit` until we reach the end. We'll start our `max_profit` at $0. As we're iterating, how do we know if we've found a new `max_profit`?

At each iteration, our `max_profit` is either:

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

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

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

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:

```ruby
def get_max_profit(stock_prices_yesterday)

    min_price = stock_prices_yesterday[0]
    max_profit = 0

    stock_prices_yesterday.each do |current_price|

        # ensure min_price is the lowest price we've seen so far
        min_price = [min_price, current_price].min

        # see what our profit would be if we bought at the
        # min price and sold at the current price
        potential_profit = current_price - min_price

        # update max_profit if we can do better
        max_profit = [max_profit, potential_profit].max
    end

    return max_profit
end
```

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. **raise 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 `max_profit` to 0 won't work...

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

```ruby
min_price = stock_prices_yesterday[0]
max_profit = stock_prices_yesterday[1] - stock_prices_yesterday[0]
```

But we have the potential for reading nil values here, if `stock_prices_yesterday` has fewer than 2 prices.

We *do* want to raise 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:

```ruby
if stock_prices_yesterday.length < 2
    raise IndexError, 'Getting a profit requires at least 2 prices'
end

min_price = stock_prices_yesterday[0]
max_profit = stock_prices_yesterday[1] - stock_prices_yesterday[0]

# etc...
```

Ok, does that work?

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

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

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

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

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

> A **greedy** algorithm iterates through the problem space taking the optimal solution "so far," until it reaches the end.
>
> The greedy approach is only optimal if the problem has "optimal substructure," which means stitching together optimal solutions to subproblems yields an optimal solution.

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 `min_price` and selling at the `current_price`
- we have a new `min_price`

To start, we initialize:

1. `min_price` as the first price of the day
2. `max_profit` 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 raised an exception instead, but returning the negative profit is cleaner, makes our function less opinionated, and ensures we don't lose information.

Ruby

```ruby
def get_max_profit(stock_prices_yesterday)

    # make sure we have at least 2 prices
    if stock_prices_yesterday.length < 2
        raise IndexError, 'Getting a profit requires at least 2 prices'
    end

    # we'll greedily update min_price and max_profit, so we initialize
    # them to the first price and the first possible profit
    min_price = stock_prices_yesterday[0]
    max_profit = stock_prices_yesterday[1] - stock_prices_yesterday[0]

    stock_prices_yesterday.each_with_index do |current_price, index|

        # skip the first time, since we already used it
        # when we initialized min_price and max_profit
        if index == 0 then next end

        # see what our profit would be if we bought at the
        # min price and sold at the current price
        potential_profit = current_price - min_price

        # update max_profit if we can do better
        max_profit = [max_profit, potential_profit].max

        # update min_price so it's always
        # the lowest price we've seen so far
        min_price  = [min_price, current_price].min
    end

    return max_profit
end
```

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

A **greedy** algorithm iterates through the problem space taking the optimal solution "so far," until it reaches the end.

> The greedy approach is only optimal if the problem has "optimal substructure," which means stitching together optimal solutions to subproblems yields an optimal solution.

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 set, 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.