You want to be able to access the *largest element* in a stack.

You've already implemented this `Stack` class:

```ruby
class Stack

    # initialize an empty array
    def initialize
        @items = []
    end

    # push a new item to the last index
    def push(item)
        @items.push(item)
    end

    # remove the last item
    def pop()
        # if the stack is empty, return nil
        # (it would also be reasonable to throw an exception)
        if @items.empty?
            return nil
        end
        return @items.pop()
    end

    # see what the last item is
    def peek()
        if @items.empty?
            return nil
        end
        return @items[-1]
    end
end
```

Use your `Stack` class to **implement a *new* class `MaxStack` with a function `get_max()` that returns the largest element in the stack.** `get_max()` should not remove the item.

Your stacks will contain only integers.

Gotchas

What if we push several items in increasing numeric order (like 1, 2, 3, 4...), so that there is a *new max* after each `push()`? What if we then `pop()` each of these items off, so that there is a *new max* after each `pop()`? Your algorithm shouldn't pay a steep cost in these edge cases.

You should be able to get a runtime of $O(1)$ for `push()`, `pop()`, and `get_max()`.

Breakdown

One lazy↴

> **Lazy evaluation** (called **short-circuit evaluation** in compiled languages) is a strategy some programming languages use to save work for the last minute or avoid unnecessary work altogether. For example, suppose we had a conditional like this:
>
> ```ruby
> if it_is_friday && it_is_raining
>     puts "board games at my place!"
> end
> ```
>
> Suppose `it_is_friday` was false. Because of the Ruby interpreter's lazy evaluation strategy, it wouldn't bother checking the value of `it_is_raining`—it knows that either way the result of our `&&` will be false, so we won't print the invitation to board game night.
>
> We can use this to our advantage. For example, suppose we have a check like this:
>
> ```ruby
> if friends['Becky'].is_free_this_friday()
>     invite_to_board_game_night(friends['Becky'])
> end
> ```
>
> What happens if 'Becky' isn't in our `friends` hash? In Ruby, we'd get `nil`, so when we try calling `is_free_this_friday()` we'll get a `NoMethodError`. (In JavaScript, we'd also get a `null` value. Python and Java would raise an error as soon as we tried looking for 'Becky' in `friends`.)
>
> Instead, we could first confirm that 'Becky' and I are still on good terms:
>
> ```ruby
> if friends.include? 'Becky' && friends['Becky'].is_free_this_friday()
>     invite_to_board_game_night(friends['Becky'])
> end
> ```
>
> This way, if 'Becky' *isn't* in `friends`, Ruby will lazily ignore the rest of the conditional and avoid trying to call `is_free_this_friday()` on `nil`!
>
> > This is all hypothetical, of course. It's not like things with Becky are weird or anything. We're totally cool. She's still in my friends hash for sure and I hope I'm still in hers and Becky if you're reading this I just want you to know you're still in my friends hash.
>
> Python's **generators** are also an example of lazy evaluation. For example, the function `range()` in Python generates a list (Python's term for an array) of numbers in a specific range:

```python
print range(1, 11)
# prints [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# (the first argument to range()
# is inclusive, and the second is exclusive)
```

This is commonly used for looping. For example, if we wanted to count to `some_high_number`, we could do this:

```python
for i in range(1, some_high_number + 1):
    print "I've eaten " + i + " cakes"
```

But this will generate a list in memory whose size is order of `some_high_number`! That could be a lot of space.

So instead, we could use a generator. It behaves like a list in that we can loop through it, but instead of building up all of its contents at once, it simply generates the *next* element right when it's needed (lazily)!

There's a generator version of `range()` in Python: `xrange()`:

```python
# much more memory efficient!
for i in xrange(1, some_high_number + 1):
    print "I've eaten " + i + " cakes"
```

In Python 3 they went ahead and made `range()` a generator, so there is no `xrange()`.

We can also take a **lazy approach**.

> **Just-in-time** and **ahead-of-time** are two different approaches for deciding when to do work.
>
> Say we're writing a function that takes in a number n between 2 and 1,000 and checks whether the number is prime.
>
> One option is to do the primality check when the function is called:
>
> ```ruby
> def prime?(n)
>     highest_possible_factor = Math.sqrt(n).floor
>     2.upto(highest_possible_factor) do |potential_factor|
>         if n % potential_factor == 0
>             return false
>         end
>     end
>     return true
> end
> ```
>
> This is a **just-in-time** approach, since we only test a number when we've received it as input. (We determine whether n is prime "just in time" to be returned to the caller.)

Another option is to generate all the primes below 1,000 once and store them in a set. Later on, when the function is called, we'll just check if n is in that set.

```ruby
require 'set'

class PrimeChecker

    def initialize
        @primes = Set.new

        2.upto(999) do |potential_prime|
            if is_prime_setup_test(potential_prime)
                @primes.add(potential_prime)
            end
        end
    end

    def is_prime_setup_test(n)
        highest_possible_factor = Math.sqrt(n).floor
        2.upto(highest_possible_factor) do |potential_factor|
            if n % potential_factor == 0
                return false
            end
        end
        return true
    end

    def prime?(n)
        return @primes.include? n
    end
end
```

Here we're taking an **ahead-of-time** approach, since we do the calculations up front before we're asked to test any specific numbers.

So, what's better: just-in-time or ahead-of-time? Ultimately, it depends on usage patterns.

If you expect `prime?()` will be called thousands of times, then a just-in-time approach will do a lot of repeat computation. But if `prime?()` is only going to be called twice, then testing all those values ahead-of-time is probably less efficient than just checking the numbers as they're requested.

> Decisions between just-in-time and ahead-of-time strategies don't just come up in code. They're common when designing systems, too.
>
> Picture this: you've finished a question on Interview Cake and triumphantly click to advance to the next question: Binary Search Tree Checker. Your browser issues a request for the question in Ruby.
>
> There are a few possibilities for what happens on our server:
>
> - One option would be to store a basic template for Binary Search Tree Checker as a starting point for any language. We'd fill in this template to generate the Ruby version when you request the page. This is a **just-in-time** approach, since we're waiting for

you to request Binary Search Tree Checker in Ruby *before* we do the work of generating the page.

- Another option would be to make separate Binary Search Tree Checker pages for every language. When you request the page in Ruby, we grab the Ruby template we made earlier and send it back. This is an **ahead-of-time** approach, since we generate complete pages *before* you send a request.

> On Interview Cake, we take an ahead-of-time approach to generating pages in different languages. This helps make each page load quickly, since we're processing our content once instead of every time someone visits a page.

in system design. For example, suppose we had a class for tracking temperatures:

Ruby

```
class TempTracker

    def initialize
        @recorded_temps = []
    end

    def record(temperature)
        @recorded_temps.push(temperature)
    end
end
```

Suppose we wanted to add a feature for getting the the highest temperature we've seen so far. We could "eagerly" keep the max up to date whenever we insert a new temperature:

Ruby

```
class TempTrackerEager

    def initialize
        @recorded_temps = []
        @max_temp = nil
    end

    def record(temperature)
        @recorded_temps.push(temperature)
        if @max_temp.nil? or temperature > @max_temp
            @max_temp = temperature
        end
    end

    def get_max
        return @max_temp
    end
end
```

Or we could lazily (or "just in time") calculate the max whenever it's requested:

```ruby
class TempTrackerLazy
    def initialize
        @recorded_temps = []
    end

    def record(temperature)
        @recorded_temps.push(temperature)
    end

    def get_max
        return @recorded_temps.max
    end
end
```

The best choice depends on how often you expect to run `get_max()`!

> Becky, I haven't hosted another board game night since the incident. I know we both said things we didn't really mean and anyway Becky just if you're reading this please know that I've been cake free for 3 whole days now and it's hard but I'm doing it for you PLEASE Becky. Please.

approach is to have `get_max()` simply walk through the stack and find the max element. This takes $O(n)$ time for each call to `get_max()`. But we can do better.

To get $O(1)$ time for `get_max()`, we could store the max integer as a member variable (call it `max`). But how would we keep it up to date?

For every `push()`, we can check to see if the item being pushed is larger than the current `max`, assigning it as our new `max` if so. But what happens when we `pop()` the current `max`? We could re-compute the current `max` by walking through our stack in $O(n)$ time. So our worst-case runtime for `pop()` would be $O(n)$. We can do better.

What if when we find a new current `max` (`new_max`), instead of overwriting the old one (`old_max`) we held onto it, so that once `new_max` was popped off our stack we would know that our `max` was back to `old_max`?

What data structure should we store our set of `max`s in? We want something where the last item we put in is the first item we get out ("last in, first out").

We can store our `max`s in another stack!

Solution

We define *two* new stacks within our `MaxStack` class—`@stack` holds all of our integers, and `@maxes_stack` holds our "maxima." We use `@maxes_stack` to keep our max up to date in constant time as we `push()` and `pop()`:

1. Whenever we `push()` a new item, we check to see if it's greater than or equal to the current max, which is at the top of `@maxes_stack`. If it is, we also `push()` it onto

`@maxes_stack`.

2. Whenever we `pop()`, we also `pop()` from the top of `maxes_stack` if the item equals the top item in `@maxes_stack`.

```ruby
class MaxStack

    def initialize()
        @stack      = Stack.new()
        @maxes_stack = Stack.new()
    end

    # Add a new item to the top of our stack. If the item is greater
    # than or equal to the last item in maxes_stack, it's
    # the new max! So we'll add it to maxes_stack.
    def push(item)
        @stack.push(item)
        if !@maxes_stack.peek() || item >= @maxes_stack.peek()
            @maxes_stack.push(item)
        end
    end

    # Remove and return the top item from our stack. If it equals
    # the top item in maxes_stack, they must have been pushed in together.
    # So we'll pop it out of maxes_stack too.
    def pop()
        item = @stack.pop()
        if item == @maxes_stack.peek()
            @maxes_stack.pop()
        end
        return item
    end

    # The last item in maxes_stack is the max item in our stack.
    def get_max()
        return @maxes_stack.peek()
    end
end
```

Complexity

$O(1)$ time for `push()`, `pop()`, and `get_max()`. $O(m)$ additional space, where m is the number of operations performed on the stack.

Notice that our time-efficient approach takes some additional space, while a lazy approach (simply walking through the stack to find the max integer whenever `get_max()` is called) took no additional space. We've traded some space efficiency for time efficiency.

What We Learned

Notice how in the solution we're *spending time* on `push()` and `pop()` so we can *save time* on `get_max()`. That's because we chose to optimize for the time cost of calls to `get_max()`.

But we could've chosen to optimize for something else. For example, if we expected we'd be running `push()` and `pop()` frequently and running `get_max()` rarely, we could have optimized for faster `push()` and `pop()` functions.

Sometimes the first step in algorithm design is *deciding what we're optimizing for*. Start by considering the expected characteristics of the input.
