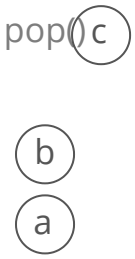
Interview Cake

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

A **stack** stores items in a last-in, first-out (LIFO) order.

Picture a pile of dirty plates in your sink. As you add more plates, you bury the old ones further down. When you take a plate off the top to wash it, you're taking the last plate you put in. "Last in, first out."

	Worst Case
space	$O(n)$
push	$O(1)$
pop	$O(1)$
peek	$O(1)$



Strengths:

- **Fast operations**. All stack operations take $O(1)$ time.

Uses:

- **The call stack** is a stack that tracks function calls in a program. When a function returns, which function do we "pop" back to? The last one that "pushed" a function call.
- **Depth-first search** (/concept/dfs) uses a stack (sometimes the call stack) to keep track of which nodes to visit next.
- **String parsing**—stacks turn out to be useful for several types of string parsing (/question/bracket-validator).

Implementation

You can implement a stack with either a linked list (/concept/linked-list) or a dynamic array (/concept/dynamic-array)—they both work pretty well:

	Stack Push	Stack Pop
Linked Lists	insert at head	remove at head
Dynamic Arrays	append	remove last element

You've already implemented this `Stack` class:

Python 2.7

```
class Stack(object):

    def __init__(self):
        """Initialize an empty stack"""
        self.items = []

    def push(self, item):
        """Push a new item onto the stack"""
        self.items.append(item)

    def pop(self):
        """Remove and return the last item"""
        # If the stack is empty, return None
        # (it would also be reasonable to throw an exception)
        if not self.items:
            return None

        return self.items.pop()

    def peek(self):
        """Return the last item without removing it"""
        if not self.items:
            return None
        return self.items[-1]
```

Use your `Stack` class to **implement a *new* class `MaxStack` with a method `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

A just-in-time

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:

Python 3.6

```
from math import sqrt

def is_prime_brute_force(n):
    highest_possible_factor = int(sqrt(n))
    for potential_factor in range(2, highest_possible_factor + 1):
        if n % potential_factor == 0:
            return False
    return True

def is_prime(n):
    return is_prime_brute_force(n)
```

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.

Python 3.6

```
from math import sqrt

def is_prime_brute_force(n):
    highest_possible_factor = int(sqrt(n))
    for potentialFactor in range(2, highest_possible_factor + 1):
        if n % potentialFactor == 0:
            return False
    return True

primes = set()

for potential_prime in range(2, 1001):
    if is_prime_brute_force(potential_prime):
        primes.add(potential_prime)

def is_prime(n):
    return n in primes
```

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 `is_prime()` will be called thousands of times, then a just-in-time approach will do a lot of repeat computation. But if `is_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 Python 3.6.
>
> 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 Python 3.6 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 Python 3.6 *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 Python 3.6, we grab the Python 3.6 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.

approach is to have `get_max()` simply walk through the stack (popping all the elements off and then pushing them back on) to 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`es 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`es 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`.

Python 3.6

```
class MaxStack(object):

    def __init__(self):
        self.stack = Stack()
        self.maxes_stack = Stack()

    def push(self, item):
        """Add a new item onto the top of our stack."""
        self.stack.push(item)

        # 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.
        if self.maxes_stack.peek() is None or item >= self.maxes_stack.peek():
            self.maxes_stack.push(item)

    def pop(self):
        """Remove and return the top item from our stack."""
        item = self.stack.pop()

        # 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.
        if item == self.maxes_stack.peek():
            self.maxes_stack.pop()

        return item

    def get_max(self):
        """The last item in maxes_stack is the max item in our stack."""
        return self.maxes_stack.peek()
```

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.

Bonus

Our solution requires $O(m)$ additional space for the second stack. But do we really need it?

Can you come up with a solution that requires $O(1)$ additional space? (It's tricky!)

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()` methods.

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