You decide to test if your oddly-mathematical heating company is fulfilling its *All-Time Max, Min, Mean and Mode Temperature Guarantee™*.

Write a class `TempTracker` with these methods:

1. `insert()`—records a new temperature
2. `get_max()`—returns the highest temp we've seen so far
3. `get_min()`—returns the lowest temp we've seen so far
4. `get_mean()`—returns the mean↴

   The **mean** of a list of values is the **average**.

   $$\text{mean} = \frac{\text{sum of all values}}{\text{number of values}}$$

   of all temps we've seen so far
5. `get_mode()`—returns a mode↴

   The **mode** of a set of values is the **number which appears the most times**.

   For example, in this set:

   ```
   1, 3, 6, 3, 1, 3
   ```

   The number 3 appears the most times, so it's the mode.

   Careful: a set may have multiple modes.

   of all temps we've seen so far

Optimize for space and time. **Favor speeding up the getter functions `get_max()`, `get_min()`, `get_mean()`, and `get_mode()` over speeding up the `insert()` function.**

`()` should return a **float**, but the rest of the getter functions can return **integers**. Temperatures will all be inserted as integers. We'll record our temperatures in Fahrenheit, so we can assume they'll all be in the range $0..110$.

If there is more than one mode, return any of the modes.

Gotchas

We can get $O(1)$ time for all functions.

We can get away with only using $O(1)$ additional space. If you're storing each temperature as it comes in, be careful! You might be taking up $O(n)$ space, where n is the number of temperatures we insert!

Are you trying to be fancy about returning multiple modes if there's a tie? Good idea, but *read the problem statement carefully*! Check out that last sentence!

> Failing to carefully read or listen to the problem statement is a *very* common mistake, and it *always* looks bad. Don't let it happen to you.

Breakdown

The first thing we want to optimize is our getter functions (per the instructions).

Our first thought might be to throw our temperatures into an array or linked list as they come in. With this method, getting the `max_temp` and `min_temp` would take $O(n)$ time. It would also cost us $O(n)$ space. But we can do better.

What if we kept track of the `max_temp` and `min_temp` *as each new number was inserted*?

That's easy enough:

```ruby
class TempTracker

    def initialize
        @min_temp = nil
        @max_temp = nil
    end

    def insert(temperature)
        if @max_temp.nil? or temperature > @max_temp
            @max_temp = temperature
        end
        if @min_temp.nil? or temperature < @min_temp
            @min_temp = temperature
        end
    end

    def get_max
        return @max_temp
    end

    def get_min
        return @min_temp
    end
end
```

This wins us $O(1)$ time for `get_max()` and `get_min()`, while keeping $O(1)$ time for `insert()` and removing the need to store all the values.

Can we do something similar for `get_mean()`?

Unlike with `min_temp` and `max_temp`, the new temp and the previous mean won't give us enough information to calculate the new mean. What other information will we need to track?

To calculate the mean of a list of values, we need to know:

- the sum of all the values
- the total number of values

So we can augment our class to keep track of the `total_numbers` and `total_sum`. Then we can compute the mean as values are inserted:

```ruby
class TempTracker

    def initialize

        # for mean
        @total_numbers = 0
        @total_sum = 0.0 # mean should be float
        @mean = nil

        # for min and max
        @min_temp = nil
        @max_temp = nil
    end

    def insert(temperature)

        # for mean
        @total_numbers += 1
        @total_sum += temperature
        @mean = @total_sum / @total_numbers

        # for min and max
        if @max_temp.nil? or temperature > @max_temp
            @max_temp = temperature
        end
        if @min_temp.nil? or temperature < @min_temp
            @min_temp = temperature
        end
    end

    def get_max
        return @max_temp
    end

    def get_min
        return @min_temp
    end

    def get_mean
        return @mean
    end
end
```

Can we do something similar for the mode? What other information will we need to track to compute the mode?

To calculate the mode, we need to know how many times each value has been inserted.

How can we track this? What data structures should we use?

Solution

We maintain the `max_temp`, `min_temp`, `mean`, and `mode` as temperatures are inserted, so that each getter function simply returns an instance variable.

To maintain the `mean` at each insert, we track the `total_numbers` and the `total_sum` of numbers inserted so far.

To maintain the `mode` at each insert, we track the total `occurrences` of each number, as well as the `max_occurrences` we've seen so far.

Ruby

```ruby
class TempTracker

    def initialize

        # for mode
        @occurrences = [0] * (111) # array of 0s at indices 0..110
        @max_occurrences = 0
        @mode = nil

        # for mean
        @total_numbers = 0
        @total_sum = 0.0 # mean should be float
        @mean = nil

        # for min and max
        @min_temp = nil
        @max_temp = nil
    end

    def insert(temperature)

        # for mode
        @occurrences[temperature] += 1
        if @occurrences[temperature] > @max_occurrences
            @mode = temperature
            @max_occurrences = @occurrences[temperature]
        end

        # for mean
        @total_numbers += 1
        @total_sum += temperature
        @mean = @total_sum / @total_numbers

        # for min and max
        if @max_temp.nil? or temperature > @max_temp
            @max_temp = temperature
        end
        if @min_temp.nil? or temperature < @min_temp
            @min_temp = temperature
        end
    end

    def get_max
        return @max_temp
    end

    def get_min
        return @min_temp
        return @min_temp
    end

    def get_mean
        return @mean
    end

    def get_mode
        return @mode
    end
end
```

We don't really *need* the getter functions since they all return instance variables. We could directly access the instance variables with `attr_reader`!

Ruby

```
# function
temp_tracker.get_mean()

# instance variable
temp_tracker.mean
```

We'll leave the getter functions in our solution because the question specifically asked for them.

But otherwise, we probably *would* use instance variables instead of functions. In Ruby we usually don't make getters if we don't *have* to, to avoid unnecessary layers of abstraction. But in Java we *would* use getters because they give us flexibility—if we wanted to change how we calculate values (for example, we might want to calculate a value 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:

Ruby

```
def fibonacci(n)
    current = 0
    next_num = 1

    n.times do
        current, next_num = next_num, current + next_num
    end

    return current
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
class Fibonacci

    def initialize
        @fibonacci_numbers = []

        @fibonacci_numbers << 0
        @fibonacci_numbers << 1

        (1000 - 2).times do
            @fibonacci_numbers << (@fibonacci_numbers[-1] + @fibonacci_numbers[-2])
        end
    end

    def fibonacci(n)
        return @fibonacci_numbers[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 `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 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.

), it won't change how other people *interact* with our class—they wouldn't have to switch from using an instance variable to using a getter method. Different languages, different conventions.

Complexity

$O(1)$ time for each function, and $O(1)$ space related to input! (Our `occurrences` array's size is bounded by our range of possible temps, in this case 0-110)

What We Learned

This question brings up a common design decision: whether to do work just-in-time or ahead-of-time.

Our first thought for this question might have been to use a **just-in-time** approach: have our `insert()` function simply put the temperature in a list, and then have our getters compute e.g. the mode just-in-time, at the moment the getter is called.

Instead, we used an **ahead-of-time** approach: have our insert function compute and store our mean, mode, max, and min *ahead of time* (that is, before they're asked for). So our getter just returns the pre-computed value in $O(1)$ time.

In this case, the ahead-of-time approach doesn't *just* speed up our getters...it also reduces our space cost. If we tried to compute each metric just-in-time, we'd need to store all of the temperatures as they come in, taking $O(n)$ space for n `insert()`s.

As an added bonus, the ahead-of-time approach didn't increase our asymptotic time cost for inserts, even though we added more work. With some cleverness (channeling some greedy↓ thinking to figure out *what we needed to store* in order to update the answer in $O(1)$ time), we were able to keep it at $O(1)$ time.

It doesn't always happen this way. Sometimes there are *trade-offs* between just-in-time and ahead-of-time. Sometimes to save time in our getters, we have to spend *more* time in our insert.

In those cases, whether we should prefer a just-in-time approach or an ahead-of-time approach is a nuanced question. Ultimately it comes down to your usage patterns. Do you expect to get more inserts than gets? Do slow inserts have a stronger negative effect on users than slow gets?

We have some more questions dealing with this stuff coming up later.

Whenever you're designing a data structure with inserts and getters, think about the advantages and disadvantages of a just-in-time approach vs an ahead-of-time approach.