Interview Cake

← course home (/table-of-contents#section_general-programming_question_temperature-tracker)

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

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

`get_mean()` should return a **float**, but the rest of the getter methods 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 methods.

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 methods (per the instructions).

Our first thought might be to throw our temperatures into a list 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:

Python 2.7

```
class TempTracker(object):

    def __init__(self):
        self.min_temp = float('inf')
        self.max_temp = float('-inf')

    def insert(self, temperature):
        if temperature > self.max_temp:
            self.max_temp = temperature
        if temperature < self.min_temp:
            self.min_temp = temperature

    def get_max(self):
        return self.max_temp

    def get_min(self):
        return self.min_temp
```

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 a mean we need to know:

- how many values there are
- the sum of all the values

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

Python 2.7

```
class TempTracker(object):

    def __init__(self):
        # For mean
        self.number_of_readings = 0
        self.total_sum = 0.0  # Mean should be float
        self.mean = None

        # For min and max
        self.min_temp = float('inf')
        self.max_temp = float('-inf')

    def insert(self, temperature):
        # For mean
        self.number_of_readings += 1
        self.total_sum += temperature
        self.mean = self.total_sum / self.number_of_readings

        # For min and max
        if temperature > self.max_temp:
            self.max_temp = temperature
        if temperature < self.min_temp:
            self.min_temp = temperature

    def get_max(self):
        return self.max_temp

    def get_min(self):
        return self.min_temp

    def get_mean(self):
        return self.mean
```

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 method simply returns an attribute.

To maintain the `mean` at each insert, we track the `number_of_readings` 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.

Python 2.7

```
class TempTracker(object):

    def __init__(self):
        # For mode
        self.occurrences = [0] * 111  # List of 0s at indices 0..110
        self.max_occurrences = 0
        self.mode = None

        # For mean
        self.number_of_readings = 0
        self.total_sum = 0.0  # Mean should be float
        self.mean = None

        # For min and max
        self.min_temp = float('inf')
        self.max_temp = float('-inf')

    def insert(self, temperature):
        # For mode
        self.occurrences[temperature] += 1
        if self.occurrences[temperature] > self.max_occurrences:
            self.mode = temperature
            self.max_occurrences = self.occurrences[temperature]

        # For mean
        self.number_of_readings += 1
        self.total_sum += temperature
        self.mean = self.total_sum / self.number_of_readings

        # For min and max
        if temperature > self.max_temp:
            self.max_temp = temperature
        if temperature < self.min_temp:
            self.min_temp = temperature

    def get_max(self):
        return self.max_temp

    def get_min(self):
        return self.min_temp
```

```
    def get_mean(self):
        return self.mean

    def get_mode(self):
        return self.mode
```

We don't really *need* the getter methods since they all return attributes. We could directly access the attributes!

Python 2.7

```
# Method
temp_tracker.get_mean()

# Attribute
temp_tracker.mean
```

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

But otherwise, we probably *would* use attributes instead of methods. In Python 2.7 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), it won't change how other people *interact* with our class—they wouldn't have to switch from using an attribute to using a getter method. Different languages, different conventions.

Complexity

$O(1)$ time for each method, and $O(1)$ space related to input! (Our `occurrences` list'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()` method simply put all of the temperatures in a list as they are recorded, 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 method 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 precomputed 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 stored all the temperatures as they came in and computed each metric just-in-time, we'd need $O(n)$ 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.

Bonus

There's at least one way to use a just-in-time approach, have $O(1)$ time for each operation, *and* keep our space cost at $O(1)$ for n inserts. How could we do that?

← course home (/table-of-contents)

Next up: Binary Numbers → (/concept/binary-numbers?course=fc1&section=bit-manipulation)