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. `getMax()`—returns the highest temp we've seen so far
3. `getMin()`—returns the lowest temp we've seen so far
4. `getMean()`—returns the mean of all temps we've seen so far
5. `getMode()`—returns a mode of all temps we've seen so far

Optimize for space and time. **Favor speeding up the getter functions (`getMax()`, `getMin()`, `getMean()`, and `getMode()`) over speeding up the `insert()` function.**

`getMean()` should return a **double**, 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 a vector or linked list as they come in. With this method, getting the `maxTemp` and `minTemp` 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 `maxTemp` and `minTemp` *as each new number was inserted*?

That's easy enough:

```cpp
class TempTracker
{
private:
    int minTemp_;
    int maxTemp_;

public:
    TempTracker() :
        minTemp_(numeric_limits<int>::max()),
        maxTemp_(numeric_limits<int>::min())
    {
    }

    void insert(int temperature)
    {
        minTemp_ = min(minTemp_, temperature);
        maxTemp_ = max(maxTemp_, temperature);
    }

    int getMax() const
    {
        return maxTemp_;
    }

    int getMin() const
    {
        return minTemp_;
    }
};
```

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

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

Unlike with `minTemp_` and `maxTemp_`, 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 `totalNumbers_` and `totalSum_`. Then we can compute the mean as values are inserted:

```cpp
class TempTracker
{
private:
    // for mean
    size_t totalNumbers_;
    double totalSum_;
    double mean_;

    // for min and max
    int minTemp_;
    int maxTemp_;

public:
    TempTracker() :
        totalNumbers_(0),
        totalSum_(0.0),
        mean_(0.0),
        minTemp_(numeric_limits<int>::max()),
        maxTemp_(numeric_limits<int>::min())
    {
    }

    void insert(int temperature)
    {
        // for mean
        ++totalNumbers_;
        totalSum_ += temperature;
        mean_ = totalSum_ / totalNumbers_;

        // for min and max
        minTemp_ = min(minTemp_, temperature);
        maxTemp_ = max(maxTemp_, temperature);
    }

    int getMax() const
    {
        return maxTemp_;
    }

    int getMin() const
```

```
    {
        return minTemp_;
    }

    double getMean() const
    {
        return 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 `maxTemp_`, `minTemp_`, `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 `totalNumbers_` and the `totalSum_` of numbers inserted so far.

To maintain the `mode_` at each insert, we track the total `occurrences_` of each number, as well as the `maxOccurrences_` we've seen so far.

C++

```cpp
class TempTracker
{
private:
    // for mode
    vector<size_t> occurrences_;
    size_t maxOccurrences_;
    int mode_;

    // for mean
    size_t totalNumbers_;
    double totalSum_;
    double mean_;

    // for min and max
    int minTemp_;
    int maxTemp_;

public:
    TempTracker() :
        occurrences_(111), // vector of 0s at indices 0..110
        maxOccurrences_(0),
        mode_(0),
        totalNumbers_(0),
        totalSum_(0.0),
        mean_(0.0),
        minTemp_(numeric_limits<int>::max()),
        maxTemp_(numeric_limits<int>::min())
    {
    }

    void insert(int temperature)
    {
        // for mode
        size_t currentTemperatureOccurrences = ++occurrences_[temperature];
        if (currentTemperatureOccurrences > maxOccurrences_) {
            mode_ = temperature;
            maxOccurrences_ = currentTemperatureOccurrences;
        }

        // for mean
```

```
        ++totalNumbers_;
        totalSum_ += temperature;
        mean_ = totalSum_ / totalNumbers_;

        // for min and max
        minTemp_ = min(minTemp_, temperature);
        maxTemp_ = max(maxTemp_, temperature);
    }

    int getMax() const
    {
        return maxTemp_;
    }

    int getMin() const
    {
        return minTemp_;
    }

    double getMean() const
    {
        return mean_;
    }

    int getMode() const
    {
        return mode_;
    }
};
```

We don't really *need* the getter functions since they all return attributes. We could directly access the attributes! (Of course, we'd have to make them public first.)

```C++
// function
tempTracker.getMean();

// attribute
tempTracker.mean_;
```

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

But otherwise, we probably *would* use attributes instead of functions. In C++ 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 need to change our logic *inside* our class, it won't change how other people *interact* with our class. Different languages, different conventions.

Complexity

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

What We Learned

This question deals a lot with just-in-time vs ahead-of-time. Or lazy vs eager. Or on-line vs off-line. There are a few names for this.

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.
