12/2/22

6.4: Array Algorithms part 1

Common Array Algorithms (**bold** ones today, others Monday)

Here are some common algorithms that you should be familiar with for the AP CS A exam:

- **Determine the minimum or maximum value in an array**
- **Compute a sum, average, or mode of array elements**
- **Search for a particular element in the array**
- **Determine if at least one element has a particular property**
- **Determine if all elements have a particular property**
- **Access all consecutive pairs of elements**
- Determine the presence or absence of duplicate elements
- Determine the number of elements meeting specific criteria
- Shift or rotate elements left or right
- Reverse the order of the elements

An exam FRQ will not be just one of these! It'll be some variation or combination... but you still need to know all of these!

Minimum and Maximum Value

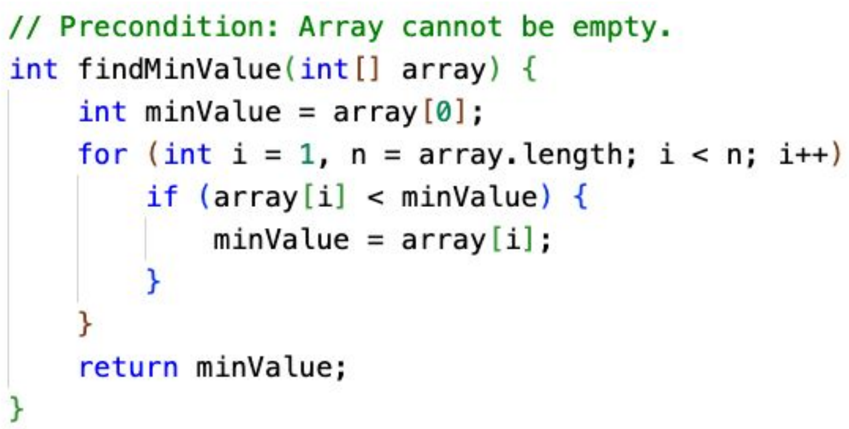
These require a "tracking value" for the smallest or largest value found so far.

25	70	9	3	15	16	19

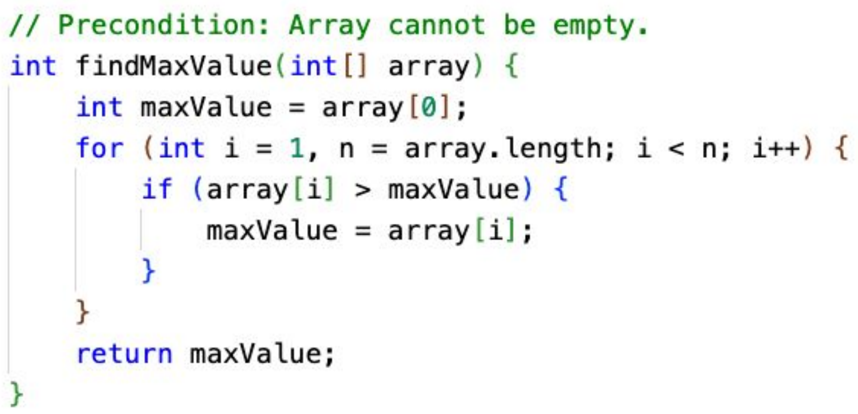
minV = 25 25 9 3 3 3 $\boxed{3}$

Minimum and Maximum Value

One trick is to "seed" the tracking value with the first element, and skip it in the loop.



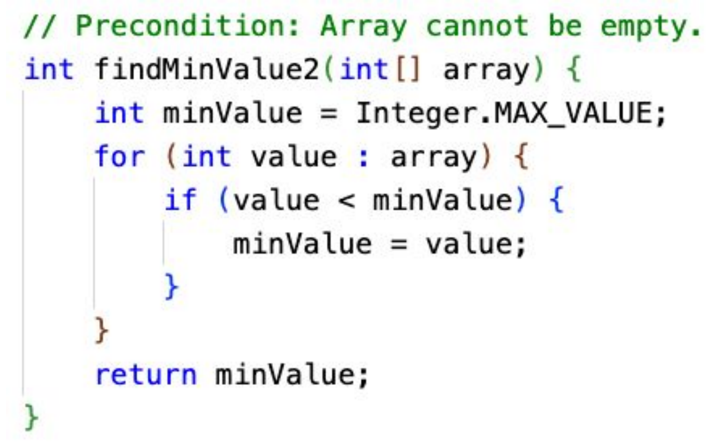
```java
// Precondition: Array cannot be empty.
int findMinValue(int[] array) {
    int minValue = array[0];
    for (int i = 1, n = array.length; i < n; i++)
        if (array[i] < minValue) {
            minValue = array[i];
        }
    }
    return minValue;
}
```



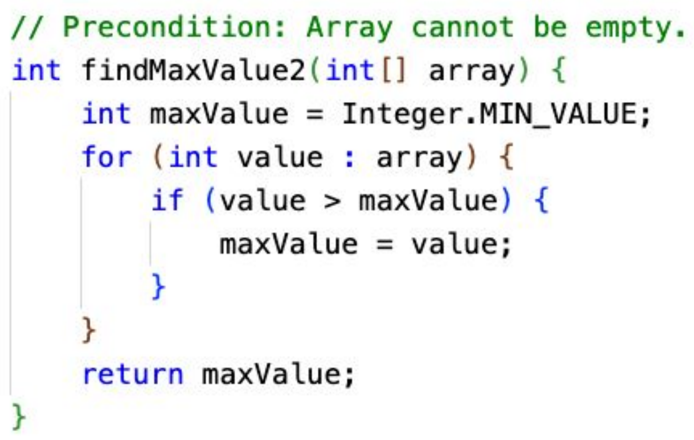
```java
// Precondition: Array cannot be empty.
int findMaxValue(int[] array) {
    int maxValue = array[0];
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] > maxValue) {
            maxValue = array[i];
        }
    }
    return maxValue;
}
```

Minimum and Maximum Value

Another way is to use the minimum/maximum value of the data type... careful which!
Or, you could use the first element here, too, and just scan it twice.



```java
// Precondition: Array cannot be empty.
int findMinValue2(int[] array) {
    int minValue = Integer.MAX_VALUE;
    for (int value : array) {
        if (value < minValue) {
            minValue = value;
        }
    }
    return minValue;
}
```



```java
// Precondition: Array cannot be empty.
int findMaxValue2(int[] array) {
    int maxValue = Integer.MIN_VALUE;
    for (int value : array) {
        if (value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
```

Minimum and Maximum Value

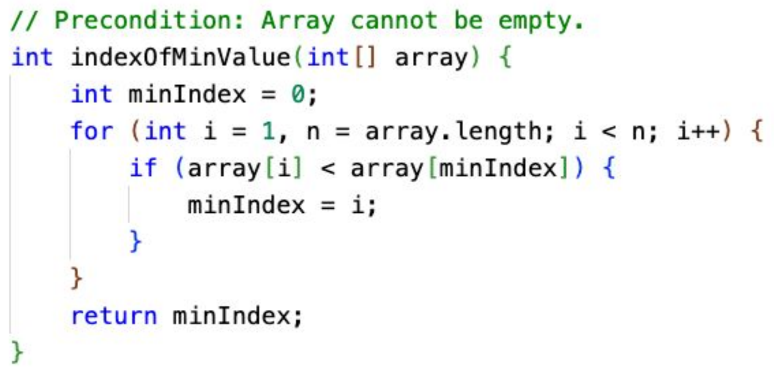
You could get rid of the precondition by returning Integer. But now callers must deal with the possibility of a **null** being returned.

```java
Integer findMinValue3(int[] array) {
    if (array.length == 0) {
        return null;
    }
    int minValue = Integer.MAX_VALUE;
    for (int value : array) {
        if (value < minValue) {
            minValue = value;
        }
    }
    return minValue;
}
```

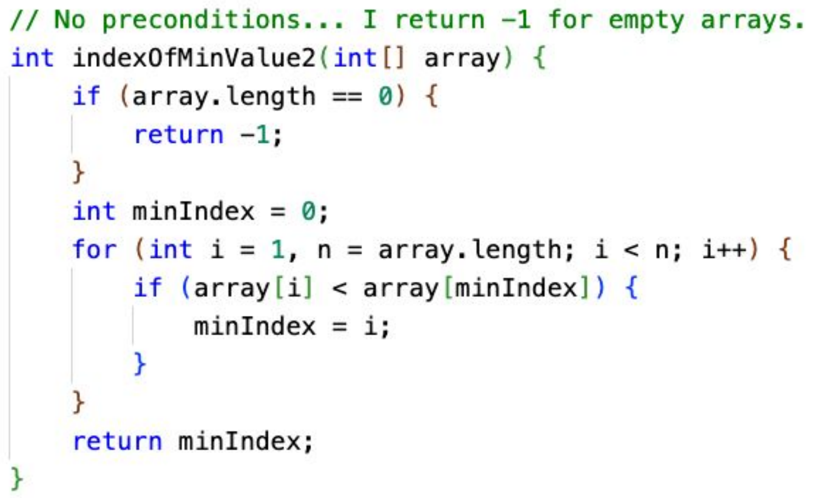
```java
Integer findMaxValue3(int[] array) {
    if (array.length == 0) {
        return null;
    }
    int maxValue = Integer.MIN_VALUE;
    for (int value : array) {
        if (value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
```

Minimum and Maximum Value

With a little modification, you could return the array index instead of the value.



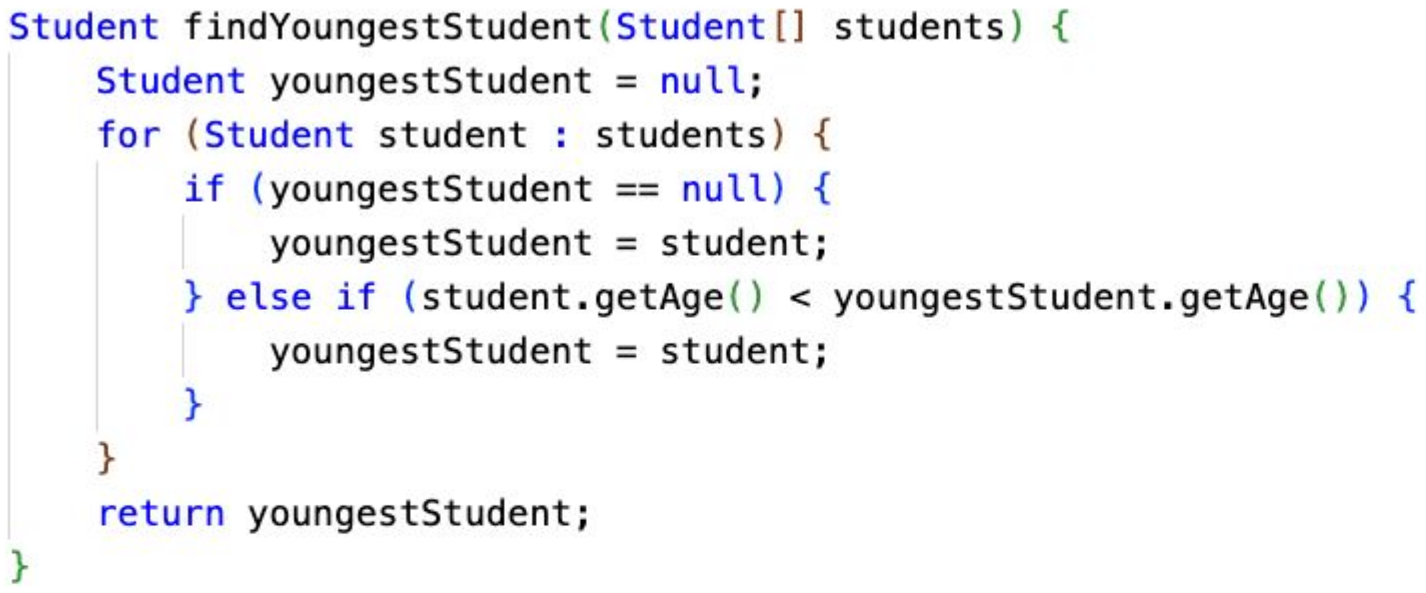
```java
// Precondition: Array cannot be empty.
int indexOfMinValue(int[] array) {
    int minIndex = 0;
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}
```



```java
// No preconditions... I return -1 for empty arrays.
int indexOfMinValue2(int[] array) {
    if (array.length == 0) {
        return -1;
    }
    int minIndex = 0;
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}
```

Minimum and Maximum Value of Objects

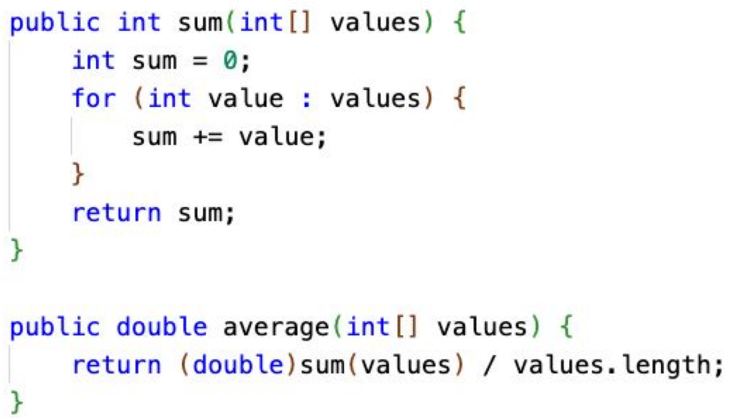
You might be dealing with an array of something other than numbers.



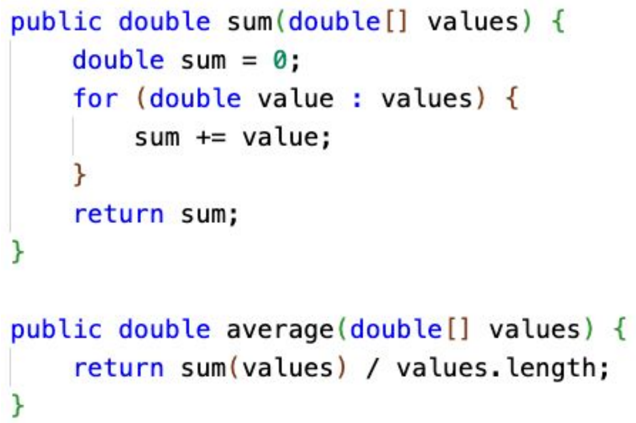
```
Student findYoungestStudent(Student[] students) {
    Student youngestStudent = null;
    for (Student student : students) {
        if (youngestStudent == null) {
            youngestStudent = student;
        } else if (student.getAge() < youngestStudent.getAge()) {
            youngestStudent = student;
        }
    }
    return youngestStudent;
}
```

Sum and Average

If dealing with ints, remember to cast to double when calculating average.
(Also known as the arithmetic mean.)



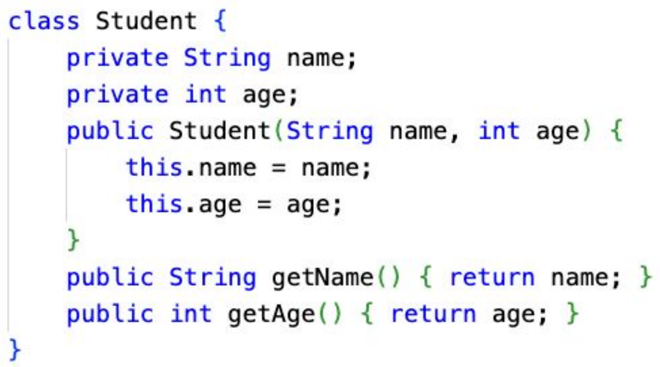
```java
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}

public double average(int[] values) {
    return (double)sum(values) / values.length;
}
```



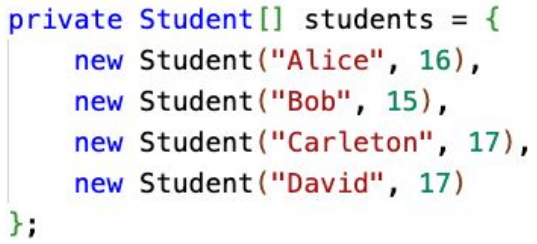
```java
public double sum(double[] values) {
    double sum = 0;
    for (double value : values) {
        sum += value;
    }
    return sum;
}

public double average(double[] values) {
    return sum(values) / values.length;
}
```

Calculations aren't always over int[] or double[]...

What if you are calculating the average age of a class of Students?



```java
class Student {
    private String name;
    private int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```



```java
private Student[] students = {
    new Student("Alice", 16),
    new Student("Bob", 15),
    new Student("Carleton", 17),
    new Student("David", 17)
};
```

Calculations aren't always over int[] or double[]...

What if you are calculating the average age of a class of Students?

```java
class Student {
    private String name;
    private int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```java
private Student[] students = {
    new Student("Alice", 16),
    new Student("Bob", 15),
    new Student("Carleton", 17),
    new Student("David", 17)
};
```

```java
public double averageStudentAge(Student[] students) {
    double sum = 0;
    for (Student student : students) {
        sum += student.getAge();
    }
    return sum / students.length;
}
```

Transforming arrays to do calculations

It can make sense to transform an Array into another Array, and then do a calculation.

```java
public double[] getStudentAges(Student[] students) {
    int count = students.length;
    double[] ages = new double[count];
    for (int i=0; i<count; i++) {
        ages[i] = students[i].getAge();
    }
    return ages;
}

public double averageStudentAge(Student[] students) {
    double sum = 0;
    for (Student student : students) {
        sum += student.getAge();
    }
    return sum / students.length;
}

public double averageStudentAge2(Student[] students) {
    return average(getStudentAges(students));
}
```

Creating the ages Array takes time and memory... but it can make sense, depending on the situation.

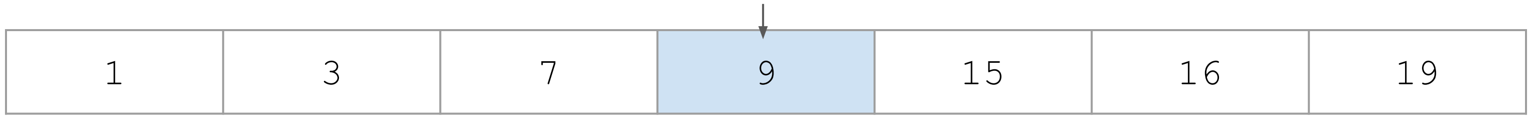
- The age data might be used more than once
- You might be interfacing with code, such as a third-party library, that doesn't know about Students
- Your math-heavy code stays in its "domain" ... it only needs to know about math, not Students, and can be reused for things other than Students.
- Performance and memory usage may not be critical

Median

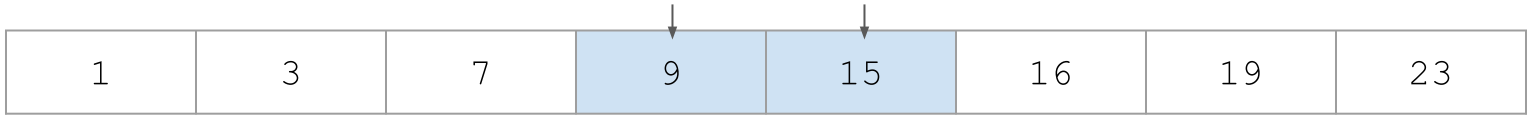
Median is defined as the "middle element" of an array.

The array needs to be sorted for it to work.

If the array has odd length, the element in the middle is returned.



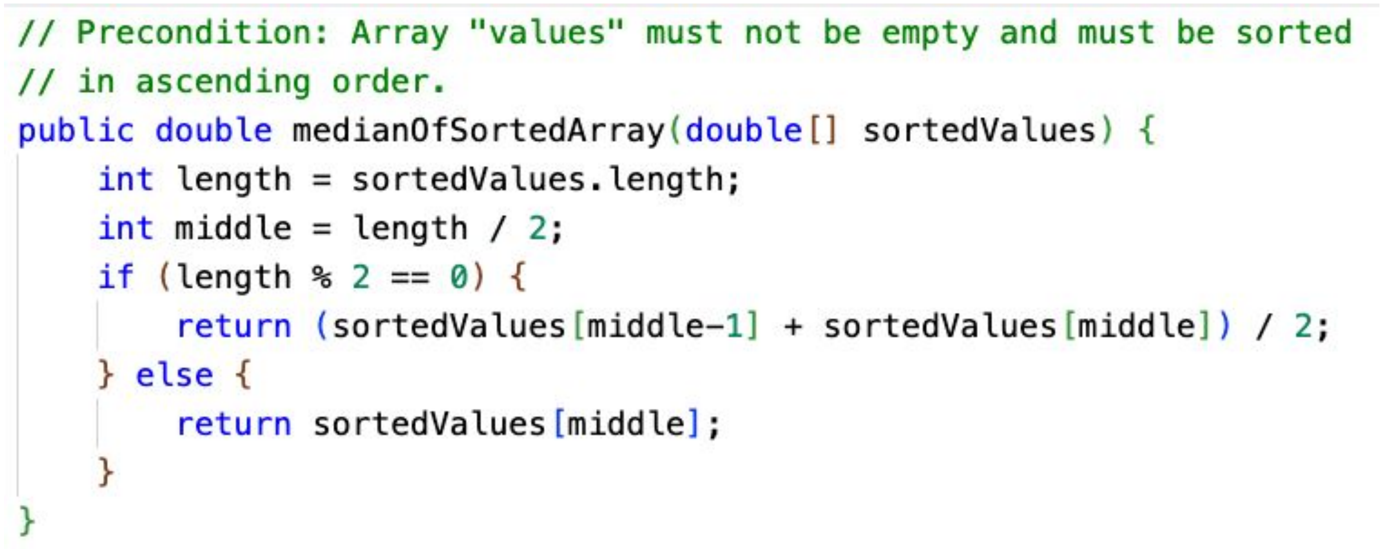
If the array is of even length, there isn't a "middle" ... so return the mathematical average of the two elements in the middle.



Median

The array needs to be sorted. We could declare a precondition!

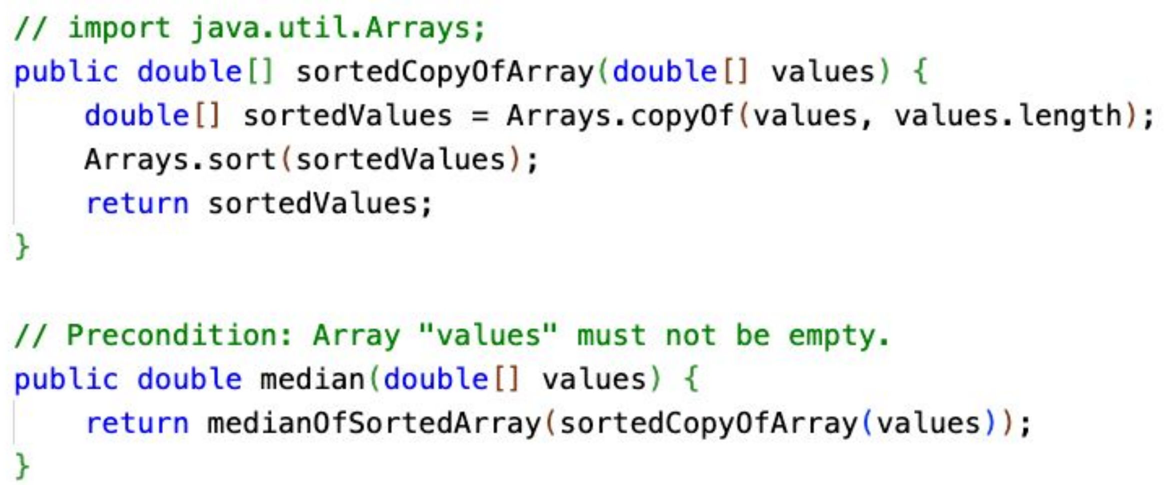
We also declare a precondition that the input array must not be empty.



```java
// Precondition: Array "values" must not be empty and must be sorted
// in ascending order.
public double medianOfSortedArray(double[] sortedValues) {
    int length = sortedValues.length;
    int middle = length / 2;
    if (length % 2 == 0) {
        return (sortedValues[middle-1] + sortedValues[middle]) / 2;
    } else {
        return sortedValues[middle];
    }
}
```

Median

We could sort the array on behalf of the caller. However, we shouldn't sort the array they passed to us by reference... that would be an unwanted side effect.



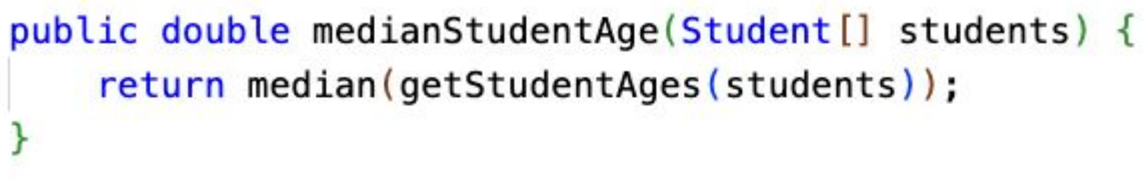
```java
// import java.util.Arrays;
public double[] sortedCopyOfArray(double[] values) {
    double[] sortedValues = Arrays.copyOf(values, values.length);
    Arrays.sort(sortedValues);
    return sortedValues;
}

// Precondition: Array "values" must not be empty.
public double median(double[] values) {
    return medianOfSortedArray(sortedCopyOfArray(values));
}
```

Median Student Age

Suppose we have an array of Students again. How do we calculate the median age?

We're able to use the tools we already built to get the student ages, and calculate the median of an array of numbers.

This is the power of **composing** methods together, and having general-purpose methods that can be applied in multiple situations. Breaking big problems into smaller problems is what CS is all about.



```java
public double medianStudentAge(Student[] students) {
    return median(getStudentAges(students));
}
```

Mode

The **mode** of an array is the value that occurs most frequently.

17	17	9	9	9	9	1	17	19	3	3	5

What is the mode of this array?

Mode

We can calculate the mode using a nested loop. For each element, count the number of occurrences of that element, and track which element has the maximum number of occurrences.

17	17	9	9	9	9	1	17	19	3	3	5

frequency (of value at current index)

3	skip	4	skip	skip	skip	1	3	1	2	skip	1

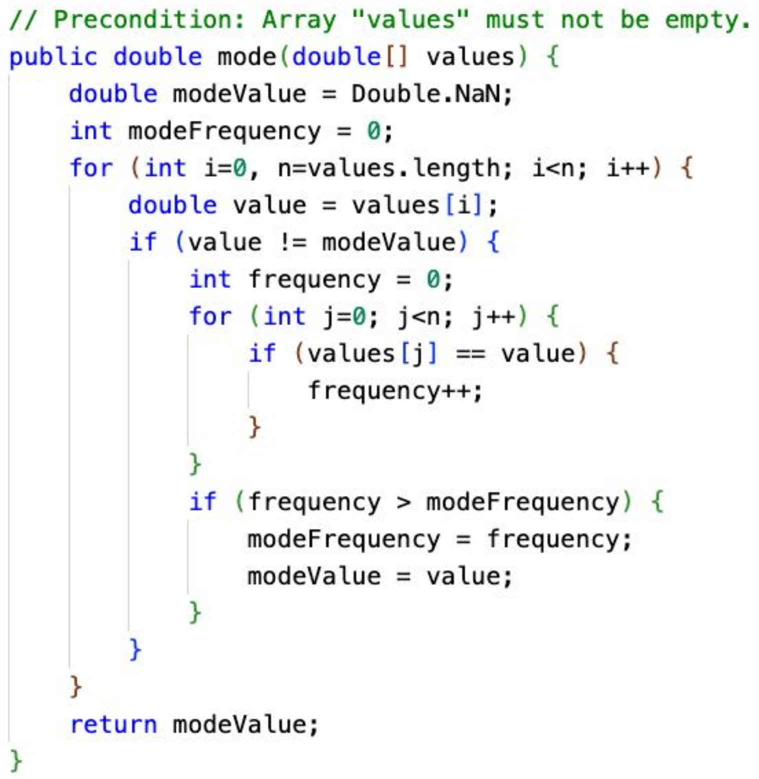
modeValue

17	17	9	9	9	9	9	9	9	9	9	**9**

modeFrequency

3	3	4	4	4	4	4	4	4	4	4	**4**

Mode



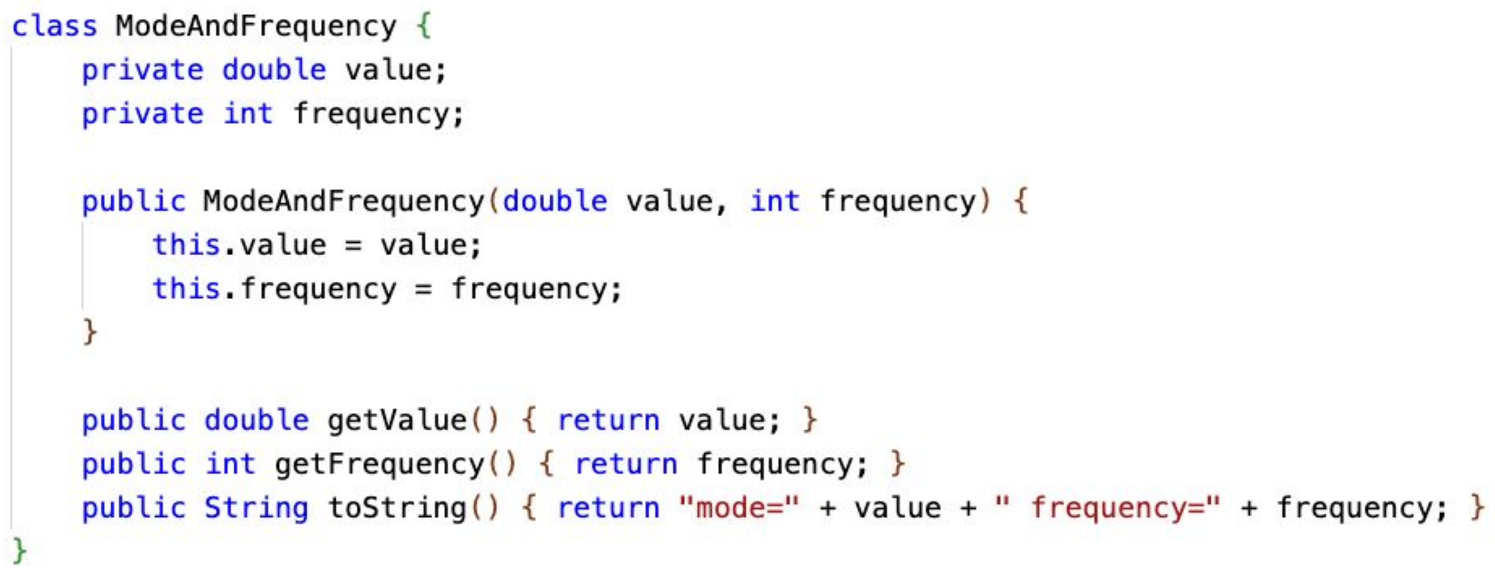
```
// Precondition: Array "values" must not be empty.
public double mode(double[] values) {
    double modeValue = Double.NaN;
    int modeFrequency = 0;
    for (int i=0, n=values.length; i<n; i++) {
        double value = values[i];
        if (value != modeValue) {
            int frequency = 0;
            for (int j=0; j<n; j++) {
                if (values[j] == value) {
                    frequency++;
                }
            }
            if (frequency > modeFrequency) {
                modeFrequency = frequency;
                modeValue = value;
            }
        }
    }
    return modeValue;
}
```

The running time of this algorithm is O(N^2).

Good? Bad?

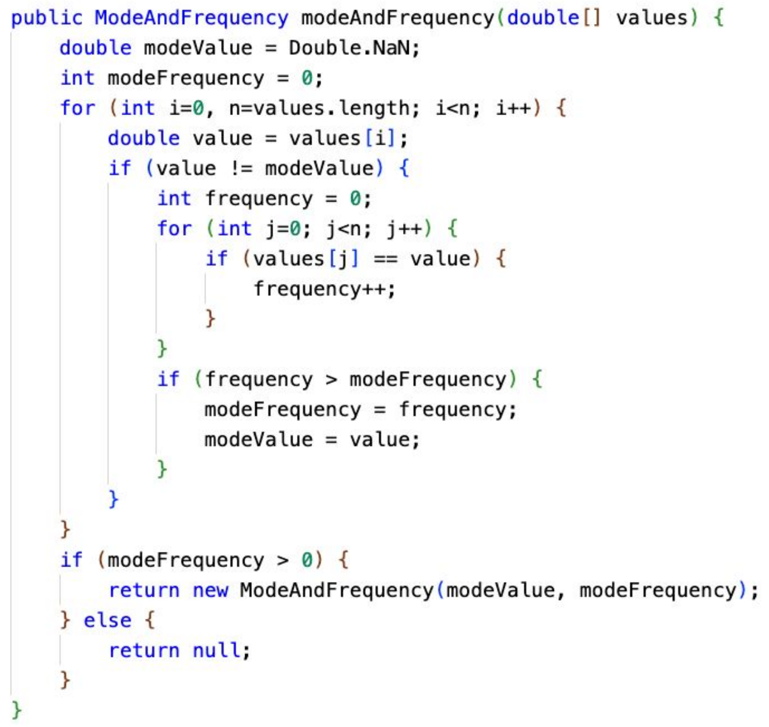
ModeAndFrequency

What if the caller wants to know both the mode value and the frequency? How do we return both values? Package it in a class.



```java
class ModeAndFrequency {
    private double value;
    private int frequency;

    public ModeAndFrequency(double value, int frequency) {
        this.value = value;
        this.frequency = frequency;
    }

    public double getValue() { return value; }
    public int getFrequency() { return frequency; }
    public String toString() { return "mode=" + value + " frequency=" + frequency; }
}
```

ModeAndFrequency



```java
public ModeAndFrequency modeAndFrequency(double[] values) {
    double modeValue = Double.NaN;
    int modeFrequency = 0;
    for (int i=0, n=values.length; i<n; i++) {
        double value = values[i];
        if (value != modeValue) {
            int frequency = 0;
            for (int j=0; j<n; j++) {
                if (values[j] == value) {
                    frequency++;
                }
            }
            if (frequency > modeFrequency) {
                modeFrequency = frequency;
                modeValue = value;
            }
        }
    }
    if (modeFrequency > 0) {
        return new ModeAndFrequency(modeValue, modeFrequency);
    } else {
        return null;
    }
}
```

Now we also can return **null** to signify that the array is empty.

Mode

What if the array is sorted? Does that make it easier to calculate the mode?

1	1	2	2	2	3	3	3	3	4	4	5

Mode of sorted array

With a sorted array, all of the repeated elements are adjacent to each other.

1	1	2	2	2	3	3	3	3	4	4	5

We can avoid a nested loop.

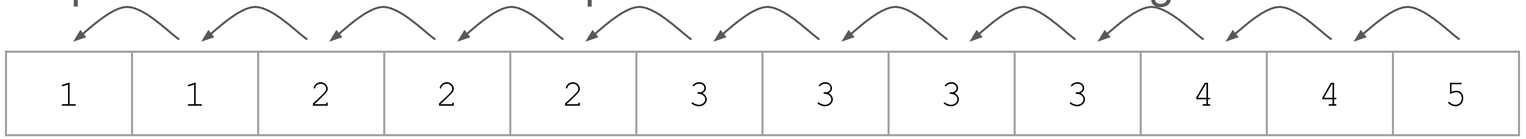
We just need to loop once, and count how many occurrences there are of each element.

We still need to track the maximum frequency, and the value associated with it.

Mode of sorted array

This algorithm is a form of **access all consecutive array elements**.

We compare each element to the previous one to see how long the "runs" are.



1	1	2	2	2	3	3	3	3	4	4	5

frequency

1	2	1	2	3	1	2	3	4	1	2	1

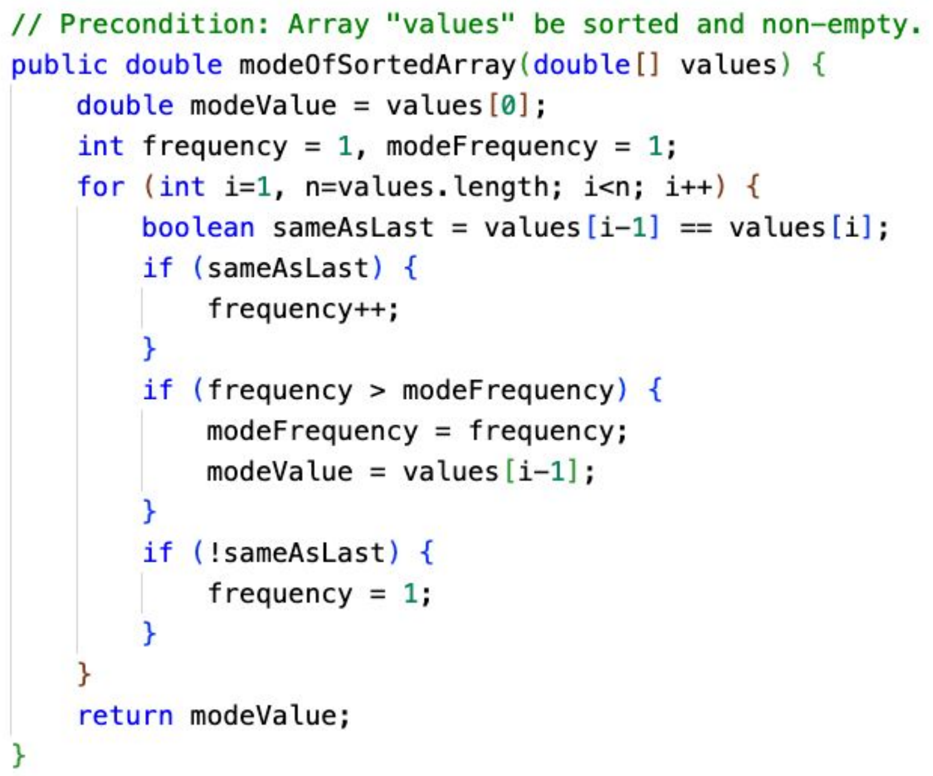
modeValue

1	1	1	1	2	2	2	2	3	3	3	3

modeFrequency

1	2	2	2	3	3	3	3	4	4	4	4

Mode of sorted array



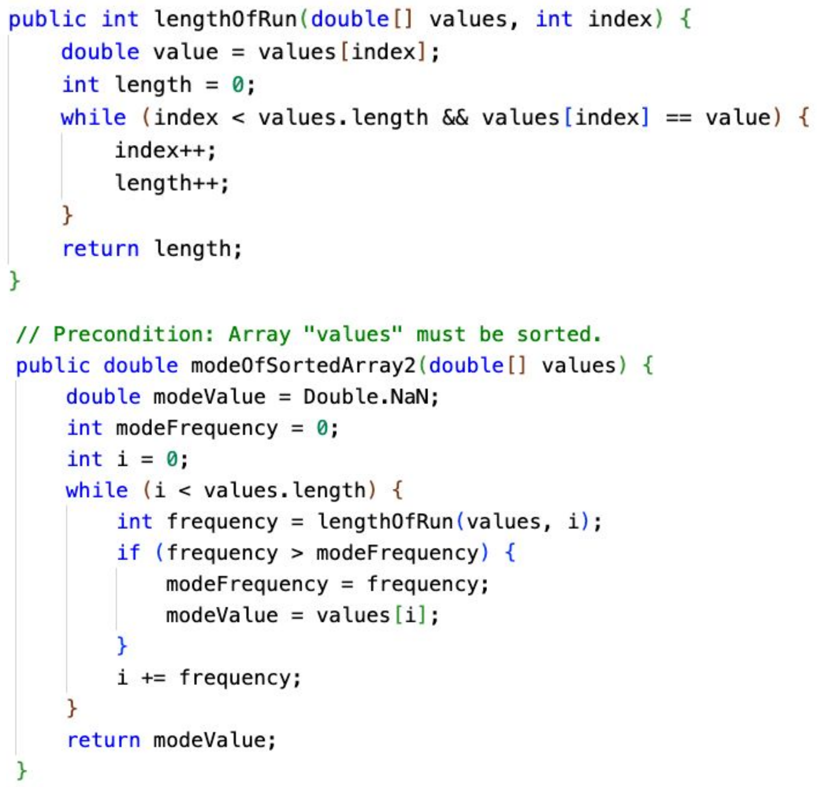
```java
// Precondition: Array "values" be sorted and non-empty.
public double modeOfSortedArray(double[] values) {
    double modeValue = values[0];
    int frequency = 1, modeFrequency = 1;
    for (int i=1, n=values.length; i<n; i++) {
        boolean sameAsLast = values[i-1] == values[i];
        if (sameAsLast) {
            frequency++;
        }
        if (frequency > modeFrequency) {
            modeFrequency = frequency;
            modeValue = values[i-1];
        }
        if (!sameAsLast) {
            frequency = 1;
        }
    }
    return modeValue;
}
```

Note how like calculating the minimum or maximum value, we can:

- Seed the tracking variables with the first element in the array
- And then, skip that element when iterating by starting the loop at i=1.

- The running time of this algorithm is O(N).
- But... the array must be sorted, and sorting takes O(N•log(N)).

Mode of sorted array



```java
public int lengthOfRun(double[] values, int index) {
    double value = values[index];
    int length = 0;
    while (index < values.length && values[index] == value) {
        index++;
        length++;
    }
    return length;
}

// Precondition: Array "values" must be sorted.
public double modeOfSortedArray2(double[] values) {
    double modeValue = Double.NaN;
    int modeFrequency = 0;
    int i = 0;
    while (i < values.length) {
        int frequency = lengthOfRun(values, i);
        if (frequency > modeFrequency) {
            modeFrequency = frequency;
            modeValue = values[i];
        }
        i += frequency;
    }
    return modeValue;
}
```

This code is a bit easier to understand than the last slide, no? It helped to break the problem up into two problems.