Lesson 16 – Algorithm Analysis & Big-O Notation

In this lesson you'll learn about analyzing algorithms to see how long they will take to run. Just in case you're not familiar with the word, an "algorithm" is just a series of steps. Each line of code is pretty much a step, so a method or a program is basically an algorithm. Of course, an algorithm could be composed of several smaller algorithms, just like a program is composed of many methods.

The most common way of describing an algorithm's runtime is through what's known as "big-O notation". O stands for "order". When big O notation is written it looks something like "O(n)", which means the program is on the order of N, or just "O of n." It looks similar to a function from algebra, because it's meant to describe your program like it's a function.

When we talk about runtimes, we're not talking about milliseconds, minutes, or any other measure of physical time. This can all change depending on the hardware that is executing the code. Instead we're describing the extent to which the execution time will change based on how many elements the program is dealing with. The number of elements is the "n" from O(n) above. As a result, this lesson is a lot "mathy-er" than previous sections, so if you get confused or stuck, please ask for help. But, once you understand it, it's not as bad as it first looks.

One last note before you get started – none of the algorithms we deal with in this course will be complicated in terms of O notation. There will only even be a few different running times total that you'll need to be able to identify.

Start by reading this blog post: https://justin.abrah.ms/computer-science/big-o-notation-explained.html in this part and the second part, when he talks about graphing things, follow along but you don't need to graph them yourself.

Part 1 exercises:

1. Describe big-o notation in your own words. (2-4 sentences is fine.)
2. Describe in one sentence what big-o notation measures
3. Write a java function that takes an integer array as input and runs in constant (O(1)) time.
4. Write a java function that takes an integer array as input and runs in linear (O(n)) time.
5. Write a java function that takes an integer array as input and runs in exponential (O(n^2)) time.
6. What kind of run case does big-o describe? (E.g. best case, worst case, average case, etc.)

Once you're done with the exercises, read the author's second blog post:
https://justin.abrah.ms/computer-science/how-to-calculate-big-o.html

Part 2 exercises:

7. How important are the coefficients before the n in your O(n) measurements? E.g. the 4 in O(4n), the 6 in O(6n^3), etc.
8. For each of the following ten code snippets below, determine and write the big-O notation for that method's running time. Answers should look like "O(1)", "O(n)", etc. If you are struggling with any given method, you can write it and add sysouts to see how the number

of sysouts grows as you change the number of elements. Doing this might also save you some time on exercise 9 (you can read that one before starting this one if you want.)

a.

```
// Exercise 8.1
public void doSomeStuff() {
    System.out.println("Do some stuff!");
}
```

b.

```
// Exercise 8.2
public int doOtherThings(int x) {
    x = x * 2;
    x = x * (x + 2);

    return x;
}
```

c.

```
// Exercise 8.3
public static long accumulator(int[] extremelyFewItems) {
    long sum = 0;

    for (long item : extremelyFewItems) {
        sum += item;
    }

    return sum;
}
```

d.

```
// Exercise 8.4
// Precondition: the input array will contain at least 15 elements
public int doThings(int[] veryData) {
    int item = 0;
    item++;

    item = veryData[2];

    return item;
}
```

e.

```
// Exercise 8.5
public boolean hasThing(int[] manyThings, int something) {
    for (int thing : manyThings) {
        if (thing == something) {
            return true;
        }
    }

    return false;
}
```

f.

```
// Exercise 8.6
public int[] squareThings(int[] maybeMoreThanZeroThings) {
    for (int thingIndex = 0; thingIndex <= maybeMoreThanZeroThings.length; thingIndex++) {
        maybeMoreThanZeroThings[thingIndex] = maybeMoreThanZeroThings[thingIndex] * maybeMoreThanZeroThings[thingIndex];
    }

    return maybeMoreThanZeroThings;
}
```

g.

```
// Exercise 8.7
public static long fastSums(int[] smallArray) {
    long sum = 0;

    for (int i = 0; i < smallArray.length; i++) {
        for (long j = 0; j < smallArray.length; j++) {
            // Super mega fast addition line
            sum += i + j;
        }
    }

    return sum;
}
```

h.

```
// Exercise 8.8
// Challenge! Be careful...
public int slowProducts(int[] bigArray) {
    int product = 0;

    for (int i = 0; i < bigArray.length; i++) {
        for (int j = 0; j < bigArray.length; j++) {
            // Super mega fast addition line
            for (int k = 0; k < bigArray[j]; k++) {
                product += k * i;
            }
        }
    }

    return product;
}
```

i.

```
// Exercise 8.9
public static long fastestOne(int[] elements) {
    long product = 0;

    for (long i = 0; i < elements.length; i++) {
        for (long j = 0; j < elements.length; j++) {
            for (long k = 0; k < elements.length; k++) {
                product = i * j * k;
            }
        }
    }

    return product;
}
```

j.

```java
// Exercise 8.10
// Challenge! Remember algebra class.
// If you're not sure, you can run this code and examine or graph the results.
public static int[] smallify(int[] numerator) {
    int smallifications = 0;

    while (numerator.length > 1) {
        // Remember that dividing integers will always result in an integers.
        // The result will be rounded down, e.g. 7 / 2 = 3.
        int halfSize = numerator.length / 2;
        numerator = new int[halfSize];

        smallifications++;
    }

    System.out.println("Smallfications: " + smallifications);

    return numerator;
}
```

9. Small project: we're going to manually run some algorithms with different O-speeds and see how they perform.
    a. You can use the built-in Java function "System.currentTimeMillis()" to get a number, in milliseconds, representing the current time. (This is returned as the variable type *long*, which is an integer but can be larger than the maximum size of *int*.) You can call this function and store its result in a variable, then run your method, then call this function again to get a new time. You can subtract the old time from the new time to get the total number of milliseconds between the two time calls. Example (I have an array of *long* variables called millisecondTimes):

```java
millisecondTimes[0] = System.currentTimeMillis();
smallify(elements);

millisecondTimes[1] = System.currentTimeMillis();
long firstRunMs = millisecondTimes[1] - millisecondTimes[0];
```

    In this example I have the subtraction right after I log the second time, but in the final version, you will want all the subtraction to be done at the end, because you're going to be taking more than one measurement. If the computer spends time doing subtraction, you don't want that to be part of your measurement.
    b. Now that you have an idea about how to measure the time, implement four methods of different O-times. If you want, you can use methods from the problem set for exercise 8, or you can write your own. Make sure that you have four different O times though – no duplicates. It's up to you how to sort out classes etc; this is not an OOP exercise so if you want to make all the methods static that's fine with me. Each method must take an integer array as input, and nothing else.
    c. Now create a method called "timeTrial" that takes:
        i. Takes an integer as a parameter
        ii. Creates an array of integers of size equal to the accepted parameter
        iii. Initialize each element of the array to a unique value

iv. Calls your four different methods while recording the runtimes for each method
v. Outputs the big-O running time of each method, along with the actual time in milliseconds, in a readable format

d. Call your timeTrial method with at least ten different numbers of elements for your array. You can do this in your main method if you want, or create a separate method. Make sure you're calling it with sufficiently different numbers of elements – e.g. the difference between one and two elements will be very small. Please cover at least four orders of magnitude, e.g. 1, 10, 100, 1000. You can see in my chart below the numbers I used and copy those if you want. I did more than ten trials. Record the results in a document or spreadsheet in an easily-readable manner. I added if statements to stop executing the slowest algorithms past a certain number of elements because otherwise it would take all day to run, which is why I have empty cells in my chart. You're free to do the same as long as they come after running times of more than 30,000 milliseconds (30 seconds.) Note that results for this exercise can vary widely, for example based on how fast your computer is and how many other processes it's running when you execute your code. (That's another reason big-O is important: it's a standard measurement.) However, if you do this exercise correctly, you will notice a huge difference in times between your algorithms, or at least between the "slow" and "fast" ones. My results:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | **Elements** | **8.1** | **8.3** | **8.7** | **8.9** |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 |
| 4 | 5 | 0 | 0 | 0 | 0 |
| 5 | 10 | 0 | 0 | 0 | 0 |
| 6 | 50 | 0 | 0 | 0 | 3 |
| 7 | 100 | 0 | 0 | 0 | 3 |
| 8 | 200 | 0 | 0 | 1 | 9 |
| 9 | 400 | 0 | 0 | 2 | 71 |
| 10 | 800 | 0 | 0 | 0 | 569 |
| 11 | 1,600 | 0 | 0 | 2 | 4,607 |
| 12 | 3,200 | 0 | 0 | 7 | 36,667 |
| 13 | 6,400 | 0 | 0 | 29 | |
| 14 | 12,800 | 0 | 0 | 129 | |
| 15 | 25,600 | 0 | 1 | 473 | |
| 16 | 51,200 | 0 | 1 | 1,916 | |
| 17 | 102,400 | 1 | 0 | 7,657 | |
| 18 | 204,800 | 0 | 1 | 30,546 | |
| 19 | 409,600 | 1 | 1 | | |
| 20 | 819,200 | 2 | 3 | **All numbers in milliseconds** | |

10. Write a paragraph or two summarizing what you've learned about big-O. Include thoughts about how significant of a difference a change in O-time makes for algorithms. Among the

O-times you've seen so far, is there any cut off between what you would consider "fast" and "slow"? Have your thoughts on exercise 7 (about coefficients) changed at all after doing exercise 9?

11. Bonus question: what happens to the robot if you write a loop that takes, say, as long to execute as method 8.9 took above, when I ran it with 800 elements? (If you don't know, it's fine – but take a guess!)