Practical Java Examples of the Big O Notation

Last modified: May 23, 2022

Written by: Orry Messer (https://www.baeldung.com/author/orry-messer)

1. Overview

In this tutorial, we'll talk about what **Big O Notation means. We'll go through a few examples to investigate its effect on the running time of your code.**

2. The Intuition of Big O Notation

We often hear the **performance of an algorithm described using Big O Notation (/cs/big-o-notation).**

The study of the performance of algorithms – or algorithmic complexity – falls into the field of algorithm analysis (https://en.wikipedia.org/wiki/Analysis_of_algorithms). Algorithm analysis answers the question of how many resources, such as disk space or time, an algorithm consumes.

We'll be looking at time as a resource. Typically, the less time an algorithm takes to complete, the better.

3. Constant Time Algorithms – *O(1)*

How does this input size of an algorithm affect its running time? **Key to understanding Big O is understanding the rates at which things can grow.** The rate in question here is **time taken per input size.**

Consider this simple piece of code:

```
int n = 1000;
System.out.println("Hey - your input is: " + n);
```

Clearly, it doesn't matter what *n* is, above. This piece of code takes a constant amount of time to run. It's not dependent on the size of n.

Similarly:

```
int n = 1000;
System.out.println("Hey - your input is: " + n);
System.out.println("Hmm.. I'm doing more stuff with: " + n);
System.out.println("And more: " + n);
```

The above example is also constant time. Even if it takes 3 times as long to run, it *doesn't depend on the size of the input, n.* We denote constant time algorithms as follows: *O(1)*. Note that *O(2)*, *O(3)* or even *O(1000)* would mean the same thing.

We don't care about exactly how long it takes to run, only that it takes constant time.

4. Logarithmic Time Algorithms – *O(log n)*

Constant time algorithms are (asymptotically) the quickest. **Logarithmic time is the next quickest.** Unfortunately, they're a bit trickier to imagine.

One common example of a logarithmic time algorithm is the binary search (https://en.wikipedia.org/wiki/Binary_search_algorithm) algorithm. To see how to implement binary search in Java, click here. (/java-binary-search)

What is important here is that the **running time grows in proportion to the logarithm of the input (in this case, log to the base 2):**

```
for (int i = 1; i < n; i = i * 2){
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

If *n* is 8, the output will be the following:

```
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 4
```

Our simple algorithm ran log(8) = 3 times.

5. Linear Time Algorithms – *O(n)*

After logarithmic time algorithms, we get the next fastest class: **linear time algorithms.**

If we say something grows linearly, we mean that it grows directly proportional to the size of its inputs.

Think of a simple for loop:

```
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

How many times does this for loop run? *n* times, of course! We don't know exactly how long it will take for this to run – and we don't worry about that.

What we do know is that the simple algorithm presented above will grow linearly with the size of its input.

We'd prefer a run time of *0.1n* than *(1000n + 1000)*, but both are still linear algorithms; they both grow directly in proportion to the size of their inputs.

Again, if the algorithm was changed to the following:

```
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
    System.out.println("Hmm.. Let's have another look at: " + i);
    System.out.println("And another: " + i);
}
```

The runtime would still be linear in the size of its input, *n*. We denote linear algorithms as follows: *O(n)*.

As with the constant time algorithms, we don't care about the specifics of the runtime. ***O(2n+1)* is the same as *O(n)***, as Big O Notation concerns itself with growth for input sizes.

6. N Log N Time Algorithms – *O(n log n)*

***n log n* is the next class of algorithms.** The running time grows in proportion to *n log n* of the input:

```
for (int i = 1; i <= n; i++){
    for(int j = 1; j < n; j = j * 2) {
        System.out.println("Hey - I'm busy looking at: " + i + " and "
+ j);
    }
}
```

For example, if the *n* is 8, then this algorithm will run *8 * log(8) = 8 * 3 = 24* times. Whether we have strict inequality or not in the for loop is irrelevant for the sake of a Big O Notation.

7. Polynomial Time Algorithms – $O(n^p)$

Next up we've got polynomial time algorithms. These algorithms are even slower than *n log n* algorithms.

The term polynomial is a general term which contains quadratic (n^2), cubic (n^3), quartic (n^4), etc. functions. **What's important to know is that $O(n^2)$ is faster than $O(n^3)$ which is faster than $O(n^4)$, etc.**

Let's have a look at a simple example of a quadratic time algorithm:

```
for (int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        System.out.println("Hey - I'm busy looking at: " + i + " and "
+ j);
    }
}
```

This algorithm will run $8^2 = 64$ times. Note, if we were to nest another for loop, this would become an $O(n^3)$ algorithm.

8. Exponential Time Algorithms – $O(k^n)$

Now we are getting into dangerous territory; these algorithms grow in proportion to some factor exponentiated by the input size.

For example, **$O(2^n)$ algorithms double with every additional input.** So, if $n = 2$, these algorithms will run four times; if $n = 3$, they will run eight times (kind of like the opposite of logarithmic time algorithms).

$O(3^n)$ algorithms triple with every additional input, $O(k^n)$ algorithms will get k times bigger with every additional input.

Let's have a look at a simple example of an $O(2^n)$ time algorithm:

```
for (int i = 1; i <= Math.pow(2, n); i++){
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

This algorithm will run $2^8 = 256$ times.

9. Factorial Time Algorithms – *O(n!)*

In most cases, this is pretty much as bad as it'll get. This class of algorithms has a run time proportional to the factorial (https://en.wikipedia.org/wiki/Factorial) of the input size.

A classic example of this is solving the traveling salesman (https://en.wikipedia.org/wiki/Travelling_salesman_problem) problem using a brute-force approach to solve it.

An explanation of the solution to the traveling salesman problem is beyond the scope of this article.

Instead, let's look at a simple *O(n!)* algorithm, as in the previous sections:

```
for (int i = 1; i <= factorial(n); i++){
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

where *factorial(n)* simply calculates n!. If n is 8, this algorithm will run *8! = 40320* times.

10. Asymptotic Functions

Big O is what is known as an *asymptotic function*. All this means, is that it concerns itself with the performance of an algorithm *at the limit* – i.e. – when lots of input is thrown at it.

Big O doesn't care about how well your algorithm does with inputs of small size. It's concerned with large inputs (think sorting a list of one million numbers vs. sorting a list of 5 numbers).

Another thing to note is that **there are other asymptotic functions.** Big Θ (theta) and Big Ω (omega) also both describes algorithms at the limit (remember, *the limit* this just means for huge inputs).

To understand the differences between these 3 important functions, we first need to know that each of Big O, Big Θ, and Big Ω describes a *set* (i.e., a collection of elements).

Here, the members of our sets are algorithms themselves:

- Big O describes the set of all algorithms that run *no worse* than a certain speed (it's an upper bound)
- Conversely, Big Ω describes the set of all algorithms that run *no better* than a certain speed (it's a lower bound)
- Finally, Big Θ describes the set of all algorithms that run *at* a certain speed (it's like equality)

The definitions we've put above are not mathematically accurate, but they will aid our understanding.

Usually, you'll hear things described using Big O, but it doesn't hurt to know about Big Θ and Big Ω.

11. Conclusion

In this article, we discussed Big O notation, and how **understanding the complexity of an algorithm can affect the running time of your code.**

A great visualization of the different complexity classes can be found here. (http://bigocheatsheet.com/)

As usual, the code snippets for this tutorial can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/algorithms-modules/algorithms-miscellaneous-3).