Arizona State University
UGTA Parrish

SER222: Data Structures & Algorithms
1/28/2018

Supplement 01.04 - Big-Oh

This supplement is intended to assist in your understanding of Big-Oh and how to figure out the Big-Oh of a formula, and code. Figure 1 is the hierarchal order of Big-Oh. This shows how fast an algorithm will preform. Such as O(1) will be faster than O(n).

$$\boxed{1 < logn < n < nlog < n^c < c^n < c^{c^n}}$$

Figure 1: Hierarchical order of Big-Oh

Finding the Big-Oh from a growth function can be done in two steps. 1) Find the largest factor and dump the other factors. 2) Remove any coefficients. What is left will be the Big-Oh. Figure 2 below shows an example.

<u>Big-Oh method:</u>

$f(n) = 3n^2 + 5n + 18$

$g(n) = 3n^2$

$g(n) = O(n^2)$

Figure 2: Big-Oh growth function example

Why can we just drop the other factors and the coefficients to find the Big-Oh? Using the proof shown below in Algorithm 1, let *k* equal the coefficients of f(n). Following the algorithm, we to see that 3n^2 will quickly overcome the other factors with any large input.

Algorithm 1 The proof for the Big-Oh method showed in Figure 2

<u>Proof:</u> $k|n| \geq f(n)$

$$f(n) = 3n^2 + 5n + 18$$
$$k|n^2| \geq f(n) = 3n^2 + 5n + 18$$
$$\text{let } k = 3 + 5 + 18$$
$$k|n^2| = 3n^2 + 5n^2 + 18n^2$$
$$3n^2 + 5n^2 + 18n^2 \geq 3n^2 + 5n + 18$$
$$3n^2 \geq 3n^2$$
$$5n^2 \geq 5n$$
$$18n^2 \geq 18$$

Looking at the last three lines in Algorithm 1, you see that *5n* and *18* will quickly be overcome by the largest factor: 3n^2. So at n goes to infinity, the only factor that will matter is the largest.

How do we get a Big-Oh expression from code? First, a line of code must be chosen to create a benchmark. Below is a simple example of this. Since the method only has code that will be executed once (no loops) this code would be O(1).

```
public static int foobar(int n) {
    System.out.println(n);    //Benchmark
}
```

Now, lets add a loop to the code. Where is a good place to set the benchmark? If *Benchmark 1* is selected, the Big-Oh would be O(*1*) which is not an effective way to see how long the code would take to execute. If *Benchmark 2* is selected, the Big-Oh would be O(n) since the loop is being included.

```
public static int foobar(int n) {
        int m = 0;                              //Benchmark 1
        for (int i = 0; i < n; i++) {
                System.out.println(i);       //Benchmark 2
        }
}
```

So how would we find code that would give O(*logn*) if a loop that increments is always going to be O(n)? Well, it depends on *how* is is being incrementing. The code below has a multiple of 4 for the loop. If a large input was given, it would traverse it at a much higher rate than if it was just a normal increment of 1. This creates the *log(n)* curve since it is much faster than just n.

```
public static int foobar(int n) {
        int m = 0;
        for (int i = 1; i < n; i *= 4) {
                System.out.println(i);              //Benchmark
        }
}
```

Nested loops will give a Big-Oh of O(n^*2*) assuming you are incrementing at a constant rate for both loops. However, applying what is shown above, a O(*nlogn*) can be shown in the code below. The inner for loop will be O(*logn*) due to the multiplier m being the incrementing factor. The outer while loop will continue to execute as a standard incrementing loop which gives O(n). With the loops combined, the O(*nlogn*) is achieved. The order does not matter in ascertaining the Big-Oh. The while loop could increment by a multiple, and the for loop increment by 1, and it will still be O(*nlogn*).

```
public static int foobar(int n) {
        int m = 15;
        while (n > m) {
                for (int i = 1; i < n; i *= m){
                        System.out.println(i);              //Benchmark
                }
                m++;
        }
}
```

The final segment of code shown below has a O(n). The *1500* in the for loop does not matter to the size of the Big-Oh. Although n must be a large input before the print statement will be executed, the code segment below is still O(n). Remember that the Big-Oh is the upper-bound and constants do not have any influence over it.

```
public static int foobar(int n) {
        for (int i = 1500; i < n; i = i++){
                System.out.println(i);              //Benchmark
        }
}
```