Big O and Asymptotic Analysis

John P. Baugh, Ph.D.

What is efficiency?

The efficiency of an algorithm is the *time* it takes to run the algorithm, and the amount of *memory* the program takes up. These are the so-called **time complexity** and **space complexity** of the algorithm.

For our purpose, we will focus on time complexity, as it is easier to analyze. In order to determine the **efficiency** of an algorithm, you may be tempted to write the code, then run it, and time it for a few different runs. While this is a valid technique for analysis, called **runtime analysis**, it has drawbacks, such as:

- If you run it on you computer, and then someone else's who has a noticeably higher-memory (or lower-memory) and/or higher CPU speed (or lower CPU speed), amongst other hardware factors, the times will not be the same, and probably won't even be in the same ballpark
- If you run the algorithm on the same computer at different times, you might get significantly different times because of other programs or factors on the system
- In order to run the algorithm, you had to have implemented it – so that takes time
    - If there are many competing algorithms, you may find you have to implement several to determine which is fastest or takes the least amount of memory

So there is a motivation for finding a technique for analyzing the efficiency of an algorithm before you take the time to implement it, and also independent of a particular device or system.

In comes the **Big O**.

Big O Analysis

The most popular technique for analyzing the complexity of algorithms is to use **Big O analysis**, also called **asymptotic analysis.** Refer to the book for a deep discussion of the topic.

We will discuss the more practical and calculation-based features that will appear on tests or assignments.

In order to perform a Big O analysis, you should know that the following are the most popular Big O **complexity categories**, in order from ***most efficient*** (least complex) to ***least efficient*** (most complex):

Category	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Linear-logarithmic
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential

You can think of each of these as a **set of algorithms**, not a single algorithm or single solution. Thus, different algorithms can be put in one of the above (typically) sets, or categories.

The value of ***n*** in the above categories indicates the **problem size**, which is often the size of a collection (such as an array) that is being processed in some way. O(1), that is, **constant time,** suggests that the time taken for processing with a particular algorithm or operation is independent of the size of the problem.

The **formal definition of Big O**, which should clarify the meaning of **asymptotic** in the alternative name, asymptotic analysis, is as follows:

Big O

We say that a function **f(n)** *is* **O(g(n))** if and only if the following holds:

$$f(n) \leq c \times g(n) \text{ for } n \geq n_0$$

Where ***n*** is the variable problem size, $\mathbf{n_0}$ is the minimum problem size at which the inequality holds, and **c** is some constant that can be multiplied by the complexity category **g(n)** to make the inequality true.

Thus, the **g(n)** will typically be one of the categories, listed above (e.g., O(1), O(log n), O(n), …)

Don't get lost in the "mathanese" of the inequality above. All it says is, your algorithm is represented by a function *f(n)*. And your algorithm is O(g(n)) ***if and only if*** it is bound from above by some constant c * g(n).

Also, **an important note** is that we aren't concerned with *all* values of n – just as *n becomes large*. So, there may be values of n for which the inequality is not true. But once it becomes true while *n* increases, it stays true. This is where the *asymptotic* part of the analysis comes from.

Intuitively, you can usually just go with the largest growing term in an expression for the Big O. So, the following expressions representing different algorithms are true:

- $3n^2 + 5n + 7$ is $O(n^2)$ because n^2 is the fastest growing term
- $3n^3 + 2$ is $O(n^3)$ because n^3 is the fastest growing term
- $15n^{10} + n^5 + 3n^2$ is $O(n^{10})$ because that is the fastest growing term
    - Note that $O(n^{10})$ isn't one of our "most common" categories

However, if you are asked to ***prove*** (write a proof) of the Big O, that is a different story. You can't just say "Looks good to me!" in mathematics. Math is more rigorous than other disciplines. Proof in a courtroom is quite different from proof in biology, or proof in mathematics.

But don't worry… even if you don't like those geometric proofs you might have done before in a geometry or trigonometry class, the proof you have to provide for Big O is actually fairly simple in most cases.

Proving Big O of an Algorithm

Remember the definition of Big O:

$$f(n) \leq c \times g(n) \text{ for } n \geq n_0$$

Identify the different parts:

- f(n) will represent the algorithm being analyzed
    - It is customary to use T(n) for time and S(n) for space when performing a specific analysis on one of these
- g(n) is the complexity category that we saw before
- c is some constant that makes the inequality true
    - Note that **c** *does not have to be unique*
- n is the size of the problem that varies as the function(s) are applied
- n_0 is the point at which the inequality becomes true
    - In other words, at some point, the c * g(n) will become and stay larger than f(n) forever, as n increases

For “Proof Problems” on tests and/or homework, you will be given the T(n), and you must prove what the Big O. The proofs are **existential** proofs, meaning you just have to **find a value for c and a value for n_0 for which the inequality is true**. That’s it. And the solution is NOT UNIQUE.

However, I have a pretty handy technique that is fast and works for all polynomial types, essentially. Let’s check out some examples.

1. $T(n) = 3n^2 + 4n + 3$
2. $T(n) = 2n^5 + 3n^2 + 5$
3. $T(n) = 10n^3 + 2$

So how do we solve these? I use something I call the **promotion technique.** You take your f(n), that is, your T(n) in our case and do the following:

1. Copy the T(n) on the left to the right of the less-than-or-equal to symbol
2. Promote each term on the right to the highest term
3. Combine like terms
4. Now you have the format that you need, from the definition of Big O, that is c * g(n)

Question 1

For $T(n) = 3n^2 + 4n + 3$:

$3n^2 + 4n + 3 <= 3n^2 + 4n^2 + 3n^2$ //promote all terms on the right to the highest term

$3n^2 + 4n + 3 <= 10n^2$ //combine like terms

So, this is in the form c * g(n), so for this proof that it is $O(n^2)$:

c = 10.

Now, finding n_0 is pretty simple.

It is true for $n_0 = 1$.

So, the Big O is proven, with sample (existential) values c = 10, and $n_0 = 1$.

Question 2

We guess $O(n^5)$ for the time complexity.

$T(n) = 2n^5 + 3n^2 + 5$

$2n^5 + 3n^2 + 5 <= 2n^5 + 3n^5 + 5n^5$

$2n^5 + 3n^2 + 5 <= 10n^5$

So, c = 10 again and $n_0 = 1$, again.

Question 3

We guess $O(n^3)$ for the time complexity.

$T(n) = 10n^3 + 2$

$10n^3 + 2 <= 10n^3 + 2n^3$

$10n^3 + 2 <= 12n^3$

So, again it's in the form c * g(n), so c = 12. And, in this format, $n_0 = 1$ again.

In fact, unless there is no trailing constant, n_0 will always be 1. The only thing that varies is c, **if you use my promotion technique.**

But, ***remember that these are NOT UNIQUE solutions***.

Now, let's see – how would functions like the above come to be in the first place?

Determining Big O from code

In addition to having to know how to do the proofs, above, you may be asked on test or homework to look at a code (or pseudocode) segment and determine the Big O. The great thing is you may not fully use the exact techniques I do, but still arrive at the same conclusion.

When determining time, even though things like multiplication take longer at the hardware or low level software hierarchy, they can be considered **basic operations,** as can determining Boolean outcome of a relational operation or something of that sort. (e.g., >=, <, !=, …)

Example 1

The numbers for the following, in the right column indicate the number of basic operations, with respect to the size of the problem. For simplicity, we assume the problem size is passed in, and is variable **n**.

```
someMethod(int n) {
    int x = 5;
    int y = 10;

    for(int i = 0; i < n; i++) {
       x += y;  //x = x + y
    }//end for
    x++;

    return x;
}
```

```
// 1
// 2    (1 for declaration, 1 for
//       init)
// 2   (same as above)
// 2 + n + n = 2n + 2
// 2 * n
//
// 1

// 1
```

2 + n + n on the header of the for loop because:

- 2 from the declaration and initialization of the variable i → it happens only once
- i < n is performed n times
- i++ is performed n times
- So, total is 2n + 2 operations with respect to n

In total, we have:

1 + 2 + 2 + 2 + 2n + 2n + 1 + 1

= 4n + 8

So, our T(n) = 4n + 8, making this O(n)

If you're asked to find the Big O, but not prove it, this is the process you can use. Note that even ***if you*** did something different (e.g., you considered int x = 5 to be 1 operation instead of 2), you will ***still*** end up with the correct big O if you followed the process. And that's fine as long as you can justify your answer.

Example 2

```
anotherMethod(int n) {                          // 1
    int value = 0;                              // 2
                                                //
    for(int i = 0; i < n; i++) {                // 2 + n + n = 2n + 2
      for(int j = 0; j < n; j++) {              // (2n + 2) * n = 2n^2 + 2n
         value++;                               // n^2
      }//end inner for
    }//end outer for


    return value;                               //1
}
```

(2n + 2) * n, is because the inner loop is 2n + 2, and for *each iteration* of the outer loop (n times), there is a full *n* iterations from the inner loop. So this nested loop is an n^2 time complexity contribution.

So we have:

$1 + 2 + 2n + 2 + 2n^2 + 2n + n^2 + 1$

$= 3n^2 + 4n + 6$

So our $O(n^2)$, and the above is our T(n)